Ganzheitliche Fehlertoleranz in eingebetteten Softwaresystemen

Der Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg
zur Erlangung des Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von

Peter Matthias Ulbrich
aus Tettnang

Als Dissertation genehmigt von der
Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung:	30. Juli 2014
Vorsitzende des Promotionsorgans:	Prof. Dr.-Ing. Marion Merklein
Gutachter:	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
	Prof. Dr. rer. nat. Jörg Kaiser

Kurzzusammenfassung

Durch die technische Entwicklung nimmt die Leistungsfähigkeit und Parallelität der Hardware stetig zu, ihre Zuverlässigkeit nimmt hingegen tendenziell ab. Im Fokus dieser Arbeit stehen die damit verbundenen, steigenden Fehlerraten für *transiente Hardwarefehler*. Diese treten zufällig auf und bewirken kurzzeitige Fehlfunktionen, welche sich ausschließlich im Feld behandeln lassen.

Der Aufbau zuverlässiger Systeme aus unzuverlässigen Komponenten erfordert grundsätzlich das Einbringen von *Redundanz*, zum Beispiel in Form von Replikation. Der naheliegende Einsatz von hardwarebasierter Redundanz scheidet jedoch häufig aus Kosten-, Gewichts- oder sonstiger Ressourcenbeschränkungen aus. An dieser Stelle bietet sich softwarebasierte Redundanz an, da diese selektiv für die Absicherung der tatsächlich sicherheitskritischen Teile der Anwendungen eingesetzt werden kann. Bestehende Ansätze erweisen sich bisher jedoch als unzuverlässig beziehungsweise lückenhaft hinsichtlich der erzielbaren Fehlererkennungsleistung. Das Problem hierbei sind *kritische Fehlerstellen* (engl. *single point of failure*) und Lücken in der Redundanz. Ein Beispiel sind die für die Replikation notwendigen Mehrheitsentscheider, welche sich ihrerseits nicht weiter replizieren lassen. Obwohl diese Stellen üblicherweise klein und kurz im Sinne der Ausführung sind, ist ihre Art und Anzahl in hohem Maße anwendungsabhängig. Eine gezielte Beeinflussung der nicht-funktionalen Eigenschaft *Zuverlässigkeit* ist somit nur schwer durch softwarebasierte Fehlertoleranz zu erreichen. Vereinfacht ausgedrückt stellt sich in diesem Fall die schon von Juvenal aufgeworfene Frage: *Wer aber wird die Wächter selbst bewachen?* (Juvenal, Satire 6, 347 f.)

Zur Lösung dieser Frage entwickelt diese Arbeit den *kombinierten Redundanzansatz* CoRed (engl. *Combined Redundancy*). Dieser ganzheitliche softwarebasierte Fehlertoleranzansatz geht von der redundanten Ausführung der Anwendung im Sinne von Prozessinkarnationen aus und beseitigt die verbliebenen Zuverlässigkeitsengpässe durch die gezielte Einbringung von Informationsredundanz in Form von arithmetischer Codierung. CoRed eliminiert als *Wächter der Wächter* die kritischen Fehlerstellen vollständig und bietet somit das Rüstzeug für die systematische Beeinflussung der Zuverlässigkeit auf Ebene des Betriebssystems. Eine vollständige experimentelle Analyse der eingesetzten arithmetischen Codierung vom Entwurf bis auf die Befehlssatzebene weist dabei die vollständige und zuverlässige Fehlererkennung und eine signifikante Verbesserung der Erkennungsleistung gegenüber vergleichbaren Ansätzen nach. Die effektive Fehlererkennung reicht dabei von den Eingängen bis zu den Ausgängen des Systems und lässt sich im Bedarfsfall auch über dessen Grenzen hinaus erweitern.

Regelungswendungen besitzen aufgrund ihrer Steuerungsfunktion und der engen Kopplung an die Umwelt typischerweise ein ausgewiesenes Schutzbedürfnis. Die praktische Anwendbarkeit des CoRed-Ansatzes wird daher am Beispiel der Fluglageregelung des unbemannten Luftfahrzeugs I4*Copter* veranschaulicht. Der Einsatz von CoRed erfolgt dabei von dessen Sensorik bis zur Aktorik und ist funktional für den Anwendungsentwickler weitgehend transparent umsetzbar. Mit der Replikation geht jedoch auch eine unvermeidbare Beeinflussung der zeitlichen Eigenschaften der Anwendung einher. Diese hat potenziell negative Auswirkungen auf die Regelungseigenschaften und erschwert die Umsetzung der Fehlertoleranz. Mit CoRed@Control untersucht diese Arbeit weitergehende Ansätze für die anwendungsgewahre Erweiterung der CoRed-Grundbausteine um eine Schnittstelle zwischen Replikation und Regelung. Diese unterstützt die Abbildung regelungstechnischer Aufgaben auf replizierbare Prozessinkarnationen des Echtzeitbetriebssystems und ermöglicht den Austausch der temporalen Eigenschaften mit der Regelungsanwendung.

Inhaltsverzeichnis

1 Einleitung und Motivation **1**
1.1 Die schwindende Zuverlässigkeit der Hardware . . . 2
1.2 Aufbau zuverlässiger Systeme aus unzuverlässiger Hardware . . . 6
1.3 Problembeschreibung und Forschungsfrage . . . 9
1.4 Lösungsansatz . . . 10
1.5 Kapitelüberblick . . . 12
1.6 Typographische Konventionen . . . 13
1.7 Zugehörige Veröffentlichungen . . . 13

I Systemebene: Fehlertoleranzbausteine **17**

2 Softwarebasierte Fehlertoleranz **19**
2.1 Terminologie und Grundlagen . . . 20
2.1.1 Fehler und deren Auswirkungen . . . 20
2.1.2 Redundanz als Mittel zur Fehlertoleranz . . . 24
2.1.3 Fehlertolerantes Verhalten . . . 27
2.2 Verwandte Arbeiten . . . 30
2.2.1 Allgemeine Betrachtung und Abgrenzung . . . 31
2.2.2 Informationsredundanz . . . 33
2.2.3 Strukturelle Redundanz und Replikation . . . 40
2.3 Zusammenfassung . . . 44

3 Der Kombinierte Redundanzansatz (CoRed) **47**
3.1 Erweiterte Problemanalyse . . . 48
3.1.1 Kritische Fehlerstellen in der Replikation . . . 49
3.1.2 Fehlererfassung und Zuverlässigkeit . . . 52
3.2 Grundannahmen . . . 53
3.2.1 Fehlerhypothese und Fehlerbereichsannahme . . . 53
3.2.2 Systemmodell . . . 54
3.3 CoRed: Ansatz und Methodik . . . 55
3.4 Eingang-zu-Ausgang Fehlererkennung . . . 58
3.4.1 Eingangsreplikation . . . 58
3.4.2 Ausgangsvergleich und Konsolidierung . . . 61
3.5 Zuverlässige Mehrheitsentscheider . . . 62
3.5.1 Arithmetische Codierung – Vergleichsoperatoren . . . 63
3.5.2 Der CoRed-Mehrheitsentscheider . . . 65
3.6 Fehlertoleranz . . . 69
3.7 Systemkomposition . . . 70
3.7.1 CoRed-Dienstschicht . . . 71

3.7.2 Abbildung auf Aufgaben des Echtzeitsystems . . . 71
3.8 Zusammenfassung . . . 72

4 Effektive Fehlererkennung – Implementierung und Evaluation 75
4.1 Erweiterte Fehlerhypothese . . . 76
4.2 Wahl der Codierungsparameter . . . 76
4.2.1 Restfehlerwahrscheinlichkeit der Codierung . . . 77
4.2.2 Auswahl der Codierungsschlüssel . . . 78
4.2.3 Auswahl der Signaturen . . . 81
4.2.4 Binäre Darstellung nicht-systematischer arithmetischer Codes . . . 82
4.2.5 Verbesserung der Erkennungsleistung bei Überbeanspruchung . . . 84
4.2.6 Resümee . . . 88
4.3 Evaluation der CoRed Implementierung . . . 88
4.3.1 Fehlerinjektion auf der Befehlssatzebene . . . 89
4.3.2 Implementierung und Versuchsaufbau . . . 90
4.3.3 Isolation der Verarbeitungsströme . . . 92
4.3.4 Prüfstein Einzelbitfehler . . . 93
4.3.5 Erweiterte Betrachtung von Mehrfachbitfehlern . . . 95
4.4 Diskussion der Ergebnisse . . . 97
4.4.1 Alternative Auswahlkriterien – Arithmetische Distanz . . . 97
4.4.2 Aussagekraft und Übertragbarkeit der Fehlerinjektion . . . 99
4.4.3 Arithmetische Codierung ganzer Anwendungen . . . 100
4.5 Zusammenfassung . . . 101

II Anwendungsebene: Replikation von Regelungsanwendungen 103

5 Fallbeispiel – I4Copter 105
5.1 Das I4*Copter* Luftfahrzeug . . . 106
5.1.1 Hardwaretechnischer Aufbau . . . 107
5.1.2 Softwaresystem . . . 110
5.2 Replikation der Fluglageregelung . . . 112
5.2.1 Umsetzung der CoRed-Dienstschicht . . . 112
5.2.2 CoRed-gesicherte Fluglageregelung . . . 113
5.2.3 Evaluation der CoRed Implementierung . . . 114
5.2.4 Kostenbetrachtung . . . 117
5.3 Erweiterung des Redundanzbereichs über Systemgrenzen . . . 118
5.3.1 Codierte Ausgabe der Stellwerte . . . 118
5.3.2 Codierte Eingabe von Messwerten und Steuernachrichten . . . 119
5.4 Zusammenfassung . . . 121

6 Weiterführende Betrachtung – CoRed@Control 123
6.1 Erweiterte Problemanalyse . . . 125
6.1.1 Diskrepanz zwischen Modellannahmen und Ausführung . . . 125
6.1.2 Auswirkungen der Replikation . . . 126
6.1.3 Verschmelzung zeitlicher Domänen . . . 127
6.1.4 Grundannahmen . . . 128
6.2 CoRed@Control: Anwendungsschnittstelle . . . 129
6.3 Modularisierung und Regelungsarchitektur . . . 130
6.3.1 Entwurfskonzept . . . 130

6.3.2 Schichtenmodell der CoRed@Control-Regelungsarchitektur 131
6.4 Modularisierung der Fluglageregelung des I4*Copters* 133
6.5 Zusammenfassung 135

7 Zusammenfassung und Ausblick **137**
7.1 Zusammenfassung 137
7.2 Beiträge dieser Arbeit 139
7.3 Schlussfolgerungen 140
7.4 Ausblick 140
7.4.1 Absicherung der Ausführungsumgebung 140
7.4.2 Werkzeugunterstützung und Automatisierung 141
7.4.3 Hydra – Hybrid-kombinierte Redundanz Architektur 142
7.4.4 Ausnutzung der inhärenten Fehlertoleranz von Regelungssystemen 143

Literaturverzeichnis **145**

Abstract **163**

Abbildungsverzeichnis

1.1 Direkte und indirekte Ionisation in einem CMOS Transistor 4
1.2 Technologiebedingte Zunahme transienter Hardwarefehler 5
1.3 Schematische Einordnung des vorgeschlagenen Lösungsansatzes 11

2.1 Fehlerausbreitung – Vom Bitfehler zum Systemversagen 21
2.2 Entwurfsentscheidungen bei der Implementierung von Fehlertoleranz 26
2.3 Die Grenzen der Fehlertoleranz: Der Redundanzbereich 27
2.4 Für den Lösungsansatz prinzipiell unzweckmäßige Fehlertoleranzmaßnahmen . . . 32
2.5 Grundlegendes Prinzip von Codierungsverfahren 33
2.6 Beispiel einer Codierungsvorschrift 34
2.7 Entwurfsalternativen für die Darstellung von Codewörtern 35
2.8 Strukturelle Redundanz in Form einer dreifach redundanten Ausführung 40

3.1 Zuverlässigkeit eines dreifach redundanten Systems 49
3.2 Entwurfsalternativen eines dreifach redundanten Systems 50
3.3 Übersicht über den CoRed-Ansatz 56
3.4 Verschiedene Arten der Eingangsreplikation in CoRed 58
3.5 Typische Abweichung der Eingangssignale eines Sensorsystems 60
3.6 Akzeptanzmaskierer für die Eingangsreplikation nichtdeterministischer Quellen . . 61
3.7 Fehlertolerante Auslegung des CoRed-Mehrheitsentscheiders 70
3.8 Übersicht über die CoRed-Dienstschicht 71

4.1 Die Restfehlerwahrscheinlichkeit im Verhältnis zur Informationsredundanz 77
4.2 Minimale Hamming-Distanz für einen 32-Bit-Code mit 16-Bit-Codierungsschlüsseln 78
4.3 Binäre Darstellung nicht-systematischer arithmetischer Codes 82
4.4 Vereinfachte Darstellung der Fehlersimulation von 1-fach bis 8-fach-Bitfehlern . . . 85
4.5 Vollständige Simulation für 32-Bit-Codes und bis zu 8-fach-Bitfehler 86
4.6 Verteilung der Restfehlerwahrscheinlichkeit in Abhängigkeit von der Fehlerzahl . . 87
4.7 Fehlerbetrachtung am Beispiel einer codierten Addition 98

5.1 CoRed Fallstudie – Der I4*Copter* Demonstrator 106
5.2 I4*Copter*-Steuergerät – Sensorik, Mikrocontroller, Aktorik 108
5.3 Schematische Darstellung der replizierten Fluglageregelung des I4*Copters* 113
5.4 Versuchsmuster einer codierten Eingabequelle in Form eines Höhensensors 120

6.1 Auswirkung der replizierten Ausführung auf die Regelgüte 124
6.2 Diskrepanz zwischen Regelungsmodell und tatsächlicher Ausführung 125
6.3 Schichtenmodell der im I4*Copter* realisierten, modularen Regelungsarchitektur . . . 131
6.4 Vergleich zwischen regulärer und modularisierter Fluglageregelung 135

7.1 Inhärente Fehlertoleranz versus zeitliche Anforderungen der Fluglageregelung . . . 142

Tabellenverzeichnis

2.1 Übersicht über die gängigsten Varianten der arithmetischen Codierung 37

4.1 Maximalfehlerzahl im Verhältnis zur Hamming-Minimaldistanz 80
4.2 IA32-Plattform: Größenvergleich einfacher versus CoRed-Mehrheitsentscheider . . 90
4.3 Evaluation des CoRed-Mehrheitsentscheider auf der Befehlssatzebene 93
4.4 Überprüfung der Hamming-Distanz auf der Befehlssatzebene durch Doppelbitfehler 95
4.5 Überprüfung der Restfehlerwahrscheinlichkeit auf der Befehlssatzebene 96
4.6 Erweiterte Fehlerinjektion von Mehrfachbitfehlern mit 3–5 Bit 97

5.1 TriCore-Plattform: Größenvergleich einfacher versus CoRed-Mehrheitsentscheider 115
5.2 Evaluation des CoRed-Mehrheitsentscheider im I4*Copter* 116
5.3 Größe und Laufzeitverhalten der CoRed-Basisoperationen 117

Programmausschnitte

2.1 Fehlersymptomatik – Vom Defekt zum Fehlverhalten 20
2.2 Symptomatische Auswirkungen transienter Fehler 23
2.3 Softwarebasierte Fehlererkennung durch Replikation auf Instruktionsebene 43

3.1 Funktionen für die arithmetische Codierung und Decodierung von Daten 57
3.2 Einfacher Mehrheitsentscheider 63
3.3 Arithmetische Codierung einer Größer-Gleich-Verzweigung 64
3.4 Zuverlässiger CoRed-Mehrheitsentscheider 66
3.5 Erkennung von Kontrollflussanomalien durch den CoRed-Mehrheitsentscheider . . 68

4.1 Schwachstelle binäre Darstellung – Integrität des Coderaumes 83
4.2 Schwachstelle Laufzeitumgebung – Integrität der isolierten Ausführung 92

Abkürzungsverzeichnis

A Codierungsschlüssel (Konstante)
Super-*As* 32-Bit-Codierungsschlüssel, besten
Abb. Abbildung
ABI application binary interface
ADC analog-to-digital converter
ALU arithmetic logic unit
AN Arithmetischer Code
ANB —, mit Signatur
ANBD —, mit Signatur und Zeitstempel
AUTOSAR AUTomotive Open System ARchitecture
AVR Atmel AVR XMEGA A4 Mikrocontroller
B Statische Signatur
bzw. beziehungsweise
CiAO CiAO Betriebssystem
CISC complex instruction set computer
CMOS complementary metal oxide semiconductor
CoRed Combined Redundancy
CoRed@Control Combined Redundancy for Control Applications
Cortex-M3 ARM Cortex-M3 Mikrocontroller
CRC cyclic redundancy check
C++ Programmiersprache
D Zeitstempel oder relativer Termin
d_h Minimale Hamming-Distanz
d_a Minimale arithmetische Distanz
DMR double modular redundancy
dt. deutsch
DUE detected unrecoverable error
e maximale Ausführungszeit (WCET)
ECC error-correcting code
engl. englisch
eV Elektronenvolt
et al. et alia
EVA Eingabe-Verarbeitung-Ausgabe
FAIL* FAIL* Fehlerinjektionswerkzeug
FIT failure in time
FPGA field-programmable gate array
GCC GNU Compiler Collection
GPGPU general purpose computation on graphics processing unit
GPR general purpose register
I4*Copter* Unbemanntes Luftfahrzeug
IA32 Intel Architecture 32-Bit
IMU inertial measurement unit
KESO Multi-JVM for Deeply Embedded Systems
MMU memory management unit
MPU memory protection unit
MU Messunsicherheit
NHN Normalhöhennull
OCDS on-chip debug support
OS operating system
ω Antwortzeit
p Periode oder Wahrscheinlichkeit
ϕ Phase einer periodischen Aufgabe
p_{sdc} SDC-Fehlerwahrscheinlichkeit
PWM pulse-width modulation
Q_{crit} kritische Ladung
R Zuverlässigkeit
R_{rep} Replikatzuverlässigkeit
R_{sys} Gesamtsystemzuverlässigkeit
R_v Zuverlässigkeit Mehrheitsentscheider
RISC reduced instruction set computer
RMT redundant multithreading
RTSC Real-Time Systems Compiler
RVM redundant virtual machine
SDC silent data corruption
SMT simultaneous multithreading
SPI serial peripheral interface
SPoF single point of failure
sog. sogenannte
SoR sphere of replication
TC1796 Infineon TriCore TC1796 Mikrocontroller
TMR triple modular redundancy
TRACE32 Diagnose-Programm (Lauterbach)

UAV unmanned aerial vehicle
v Variablenbezeichner, uncodierter Wert
v_c Variablenbezeichner, codierter Wert
vgl. vergleiche
VM virtual machine
VMM virtual machine monitor
WCET worst case execution time
WLAN wireless local area network
z. B. zum Beispiel
$\bigcirc$ Arithmetische Operation, uncodiert
$\bigcirc_c$ Arithmetische Operation, codiert

1 Einleitung und Motivation

Seit den ersten Ansätzen zur Steuerung des Stapelbetriebs in den 1950er Jahren ist die Systemsoftware eng mit dem technologischen Fortschritt gekoppelt [Han01]. Die kontinuierliche Entwicklung der Hardware und die dadurch entstehenden Möglichkeiten und Herausforderungen stellen eine der Triebfedern der Forschung und Entwicklung in diesem Bereich dar. So folgten der rasanten Leistungssteigerung der Rechensysteme ebenso fortschrittliche Konzepte im Bereich der Systemsoftware wie Mehrprozessbetrieb und Nebenläufigkeit, die Einbeziehung des Betriebsmittels Energie oder auch, mit dem Aufkommen leistungsfähiger Kommunikationsnetzwerke, die Unterstützung verteilt arbeitender Systeme – um nur einige Beispiele zu nennen.

Die Systemsoftware bildet die Schnittstelle zwischen dem eigentlichen Rechensystem und der Anwendungssoftware. Zu ihren zentralen Aufgaben zählen dabei die Abstraktion von den Implementierungsdetails der Hardware, die Verwaltung der Betriebsmittel sowie die Bereitstellung von Strukturmitteln für die logische Aufteilung der Anwendungssoftware. Zu diesem Zweck stellt die Systemsoftware dem Anwendungsentwickler abstraktere Schnittstellen zur Verfügung, welche ihrerseits wiederum häufig als abstrakte virtuelle Maschinen gesehen werden. Die Abstraktion findet üblicherweise in dedizierten Schichten [Tan06] statt, zum Beispiel beginnend auf der Ebene der Hardware- beziehungsweise Befehlssatzarchitektur über das Betriebssystem bis hin zu einer Zahl von darauf aufbauenden *Dienstschichten* (engl. *middleware*). Das Ziel jeder Schicht ist es, Komplexität zu verbergen und auf oberster Ebene im Sinne des Anwendungsentwicklers eine optimale Abstraktion zu erreichen. Als Vermittler zwischen Anwendung und Rechensystem bildet die Systemsoftware dabei nicht nur die funktionalen, sondern auch die nicht-funktionalen Eigenschaften der Hardware auf die Anforderungen der Software ab. Zu diesen zählen zum Beispiel die *Kosten-* und *Energieeffizienz*, *Rechtzeitigkeit* oder *Zuverlässigkeit*.

Diese Dissertation beschäftigt sich im Speziellen mit der nicht-funktionalen Eigenschaft Zuverlässigkeit aus dem Blickwinkel der Systemsoftware und den Herausforderungen, die sich aus dem technologiebedingten Schwinden dieser Zuverlässigkeit auf Ebene der Hardware ergeben. Im Fokus stehen dabei sogenannte *transiente Hardwarefehler* (engl. *soft errors*), die im Folgenden für ein besseres Verständnis zunächst näher eingeführt werden. Der zweite wichtige Aspekt dieser Arbeit ist die optimale Unterstützung der Anwendung durch die Bereitstellung geeigneter Strukturelemente und Schnittstellen für deren Abbildung auf die Systemsoftware. Daher richtet

sich der Blick anschließend auf die Klasse der *sicherheitskritischen Regelungsanwendungen*, welche repräsentativ für hohe Anforderungen sowohl an die Zuverlässigkeit als auch an die *Rechtzeitigkeit* stehen. Dieses Kapitel skizziert die eigentliche Problemstellung und den verfolgten Lösungsansatz für eine gezielte Beeinflussung der Zuverlässigkeit von sicherheitskritischen Regelungsanwendungen durch die Systemsoftware. Das Kapitel schließt mit einem Überblick über den weiteren Aufbau dieser Arbeit sowie mit einer Zusammenfassung der zugehörigen Veröffentlichungen.

1.1 Die schwindende Zuverlässigkeit der Hardware

Die korrekte Ausführung von Programmcode durch das darunterliegende Rechensystem, genauer gesagt dessen Prozessoren, ist eine übliche Grundannahme der Softwaretechnik. Gleichwohl stellen transiente Fehler eine zunehmende und in vielen Fällen unterschätzte Bedrohung für die Zuverlässigkeit sicherheitskritischer Systeme dar. Einen Einblick in die Ursachen, Hintergründe und Tendenzen bieten dieses Kapitel 1.1 sowie der Exkurs zur *kosmischen Strahlung* auf Seite 4. Die beiden Exzerpte auf den Seiten 3 und 6 erlauben jedoch eine direkte Abkürzung zu den technischen Gegenmaßnahmen in Kapitel 1.2 ab Seite 6.

Hardwarefehler – Grundlagen

Die Halbleitertechnik ist, wie jedes technische System, in der Praxis nicht frei von Fehlern und bietet entsprechend keine absolute Zuverlässigkeit. Die Hersteller geben daher üblicherweise eine *Ausfallrate*[1] (*FIT*) (engl. *failure in time*) für ihre Produkte an. Eine häufig gewählte Darstellungsform stellt die sogenannte *Badewannenkurve* [Wil02] dar. Sie beschreibt die Zuverlässigkeit eines Systems über dessen Lebenszyklus: Im Zuge der Inbetriebnahme wirken sich zunächst gehäuft immanente beziehungsweise fertigungsbedingte Fehler aus, bevor die Fehlerrate auf ein niedrigeres, relativ konstantes Niveau abfällt. Zum Ende der Lebensdauer kommt es wiederum gehäuft zu Abnutzungserscheinungen an den Halbleiterbauelementen und alterungsbedingten Fehlern.

Grundsätzlich unterscheidet man *permanente*, *intermittierende* und *transiente Hardwarefehler* gemäß der Art ihres Auftretens. Beispiele für permanente Fehler in der heute vorherrschenden *complementary metal oxide semiconductor (CMOS)* Technologie sind unter anderem Verunreinigungen des Leitermaterials und daraus resultierende Elektromigration[2] oder Leckströme durch fehlerhafte Oxidschichten der Transistoren [Sta02]. Intermittierende Hardwarefehler treten abhängig von äußeren Einflüssen, wie zum Beispiel der Temperatur, auf und sind häufig Vorboten eines permanenten Hardwarefehlers. Die Hersteller begegnen diesen systematischen Fehlern beispielsweise durch Testen, Optimierung des Entwurfs und der Produktion sowie dem Einbrennen[3] der Schaltungen vor der Auslieferung. [Brü05, SH04]

Transiente Hardwarefehler (soft errors)

Im Fokus dieser Arbeit stehen die *transienten Hardwarefehler* (engl. *soft errors*). Sie treten zufällig auf und ziehen keine irreversiblen Defekte nach sich, sondern bewirken vielmehr eine kurzzeitige Fehlfunktion. Unabhängig von der konkreten Ursache manifestieren sich transiente Fehler aus Sicht der Software üblicherweise in Form von Einfach- oder Mehrfach-*Bitfehlern* (engl. *single event upset* und *multi-cell upset*). Transiente Hardwarefehler unterscheiden sich von den zuvor genannten systematischen Fehlerursachen in zwei wesentlichen Punkten: Zum einen treten sie im Feld

[1] Die Ausfallrate λ gibt die Anzahl der Fehler in 10^9 Stunden an.
[2] Der Materialtransport in einem festen Leiter verursacht durch den elektrischen Strom.
[3] Der extensive Betrieb an der Leistungsgrenze für die Dauer der Frühfehlerphase.

auf und müssen auch dort behandelt werden, entsprechend erfordern sie explizite Maßnahmen zur Fehlererkennung und Fehlerkorrektur. Zum anderen steigt die Fehlerrate technologiebedingt immer weiter an. [Bor05, SH04, ZP04]

Die primäre Ursache für transiente Hardwarefehler ist ionisierende Strahlung, wie zum Beispiel Neutronen- oder Alphastrahlung, welche zu einer Ladungsverschiebung innerhalb des Halbleitermaterials führen kann. Ist diese Verschiebung hinreichend groß, kommt es zu einer Inversion des Logikzustandes innerhalb einer Schaltung und damit zu einer Manifestierung des Fehlers. [Bau05, Brü05, SH04]

Erstmals wurde 1978 von Intel über transiente Hardwarefehler, induziert durch Alphastrahlung, berichtet [MW79]. Der Auslöser war eine radioaktive Kontamination der keramischen Chipgehäuse mit Uran. Durch die räumliche Nähe konnten die hiervon ausgehenden Alphateilchen direkt auf die Halbleiterbauelemente wirken. In ihrer näheren Untersuchung des Problems führten May und Woods [MW79] das zentrale Konzept der *kritischen Ladung* (Q_{crit}) ein. Dieses gibt an, wie groß die induzierte Ladungsverschiebung für eine Fehlfunktion der Schaltung sein muss. Bereits im selben Jahr postulierten Ziegler und Lanford [ZL79] ähnliche Auswirkungen durch *kosmische Strahlung* (vgl. Exkurs, Seite 4), wobei sie von einer Zunahme der Fehlerrate mit der Höhe ausgingen. Der erste Nachweis von transienten Hardwarefehlern durch kosmische Strahlung in Standardcomputerbauteilen erfolgte 1984 durch interne Untersuchungen von IBM [ZCM+96]. Mit dem Einzug entsprechender Bauteile in der Luft- und Raumfahrt folgten weitere Untersuchungen, welche die Annahmen von Ziegler und Lanford bestätigten [TN93]. Im selben Maß konnten die Auswirkungen der kosmischen Strahlung auf größere (bodengebundene) Systeme nachgewiesen werden [Nor96].

Im Allgemeinen können sich Variationen und Fehler in der Produktion ebenfalls in einem sporadischen Fehlverhalten äußern. Zu den Hauptursachen zählen unter anderem Fertigungsstreuungen der Transistoren, temporäre Materialdefekte durch thermale Wechselbeanspruchung oder eine instabile Spannungsversorgung. Obwohl grundsätzlich produktionsbedingt und damit auch den systematischen Fehler zurechenbar, sind diese Effekte für die Hersteller nur sehr schwer beherrschbar, da zum Beispiel die Fertigungstoleranzen bei Nanolithografie inzwischen bei wenigen Atomlagen liegen. [Bor05, SH04]

Eine weitere Ursache für transiente Hardwarefehler sind Störimpulse. Sie entstehen durch elektromagnetische Wechselfelder und führen zu Rauschen oder Übersprechen von Signalen auf Bussen und Datenleitungen. Da diese Art der Fehler prinzipiell durch eine ausreichende elektromagnetische Abschirmung vermeidbar ist, stellt sie ebenfalls eine Grauzone dar. Im Unterschied zu den anderen Fehlerursachen wirken Störimpulse nicht direkt auf die Halbleiterbauelemente, sondern beeinflussen typischerweise die Kommunikationsleitungen, welche diese verbinden.

Begriffsbildung: Transienter Fehler

Im weiteren Verlauf wird der Begriff *transienter Fehler* synonym für die Klasse der transienten Hardwarefehler genutzt, die sich auf Ebene der Hardware (Datenspeicher, Rechenwerke oder Busse) als Bitfehler manifestieren und im Feld behandelt werden müssen.

Fehlerraten transienter Fehler – Entwicklung und Tendenzen

Transiente Fehler stellen seit jeher eine Herausforderung bei der Entwicklung besonders sicherheitskritischer Systeme dar. Dies gilt vor allem in Bereichen, in denen die Fehlerraten umweltbedingt erhöht sind. So steigt die Strahlung und proportional dazu auch die Ausfallrate mit der Höhe an (siehe Abbildung 1.2, links). In der für Verkehrsflugzeuge üblichen Flughöhe von rund

Exkurs: Kosmische Strahlung[a]

Die *kosmische Strahlung*, Hauptursache für transiente Fehler in CMOS-Bauelementen, verändert ihre Zusammensetzung und Wirkungsweise auf dem Weg durch die Atmosphäre:

Primäre kosmische Strahlung

Die *primäre kosmische Strahlung* ist eine hochenergetische Teilchenstrahlung aus dem Weltall, die ihren Ursprung in kosmischen Ereignissen (z. B. Supernovae) und Objekten (z. B. Sternen) hat. Sie besteht zum größten Teil aus Protonen mit einer hohen Teilchenenergie[b] von circa 1 GeV. Die Flussdichte beträgt rund 36 000 $^{\text{Partikel}}/_{\text{cm}^2\,\text{h}}$.

Sekundäre kosmische Strahlung

Ab einer Höhe von rund 20 km kollidieren die Teilchen der Primärstrahlung mit den in der Atmosphäre enthaltenen Atomen. Aus einem Proton entsteht dabei in einer Kaskade Millionen Teilchen der *sekundären kosmischen Strahlung*. Zu ihnen gehören hauptsächlich Neutronen, Pionen und Myonen. Die Halbwertszeit der Pionen und Myonen liegt im Bereich von Mikro- beziehungsweise Nanosekunden, weshalb diese sehr schnell zerfallen. Demgegenüber liegt die Halbwertszeit der Neutronen bei annähernd 15 Minuten, wodurch die Kaskade fortgesetzt wird bis die Teilchenenergie nicht mehr für eine weitere Spaltung ausreicht. Der *Neutronenfluss* variiert dadurch in Abhängigkeit von der Höhe wie in Abbildung 1.2 (links) dargestellt. Die folgende Gleichung liefert näherungsweise die Zunahme der Neutronenflussdichte, wobei H die Höhe in Kilometer ist:

$$\text{Flusszunahme über Meeresspiegel} = e^{\left(\frac{119{,}685 \cdot H - 4{,}585 \cdot H^2}{136}\right)}$$

Neben der Höhe spielt noch die Stärke des Erdmagnetfelds in Abhängigkeit des Längen- beziehungsweise Breitengrads eine Rolle. Entsprechend genauere Berechnungsmethoden bilden gleichwohl die Basis für korrespondierende Industriestandards. [JED06]

Terrestrische kosmische Strahlung

Unter der *terrestrischen kosmischen Strahlung* versteht man die Teilchenstrahlung, die tatsächlich auf der Erdoberfläche auftrifft. An dieser Stelle ist der Anteil der Primärstrahlung auf unter 1 % gesunken und ein Großteil der Sekundärteilchen ist entweder zerfallen oder hat seine kinetische Energie verloren. Aktuelle Messungen von Gordon et al. [GGR+04] zeigen einen Neutronenfluss von durchschnittlich 20 $^{\text{Partikel}}/_{\text{cm}^2\,\text{h}}$ (Teilchenenergie ≥ 10 MeV).

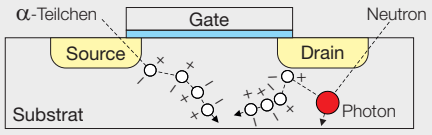


Abbildung 1.1: Schematischer Vergleich von direkter und indirekter Ionisation in einem CMOS Transistor.

Auswirkung auf die Halbleiterbauelemente

In erster Näherung ist die Ausfallrate für einen CMOS-Schaltkreis proportional zum Neutronenfluss. Neutronen wirken, wie in Abbildung 1.1 dargestellt, nur indirekt ionisierend. Die Ionisation erfolgt mittelbar über Kernreaktionen oder Streuprozesse an Atomkernen – vergleichbar den Kollisionen in der Atmosphäre. Werden die hierbei entstehenden Elektronen, Mesonen und positiven Ionen durch das Halbleitermaterial abgebremst, verlieren diese schrittweise ihre Energie und erzeugen dabei *Elektron-Loch-Paare*[c]. Die hierbei entstehende Ladung verteilt sich entlang des Eintrittspfads und bestimmt sich aus der kinetischen Energie des Partikels und dem Energieverlust pro Weglängeneinheit, dem sogenannten *Bremsvermögen* (engl. *stopping power*) des Materials. Ein Elektron-Loch-Paar in Silizium erfordert zum Beispiel etwa 3,6 eV Energie, das Bremsvermögen für Alphateilchen liegt bei circa 100 $^{\text{keV}}/_{\mu\text{m}}$ woraus sich $2{,}8 \cdot 10^4$ $^{\text{Elektron-Loch-Paare}}/_{\mu\text{m}}$ und eine Ladung von 4,5 $^{\text{fC}}/_{\mu\text{m}}$ ergeben. Daher reicht für heute gängige Werte für Q_{crit} von unter 1 fC bereits eine sehr kleine Teilchenenergie von circa 22 keV aus, um ein Fehlverhalten hervorzurufen.

[a] Vergleiche [UBD02, S. 363ff], [Bau05] und [ZP04].
[b] 1 eV ist die kinetische Energie eines Elektrons, welches eine Beschleunigungsspannung von 1 Volt durchläuft.
[c] Ein Paar aus einem Elektron sowie einem Defektelektron, welches durch Energieabsorption angeregt wurde.

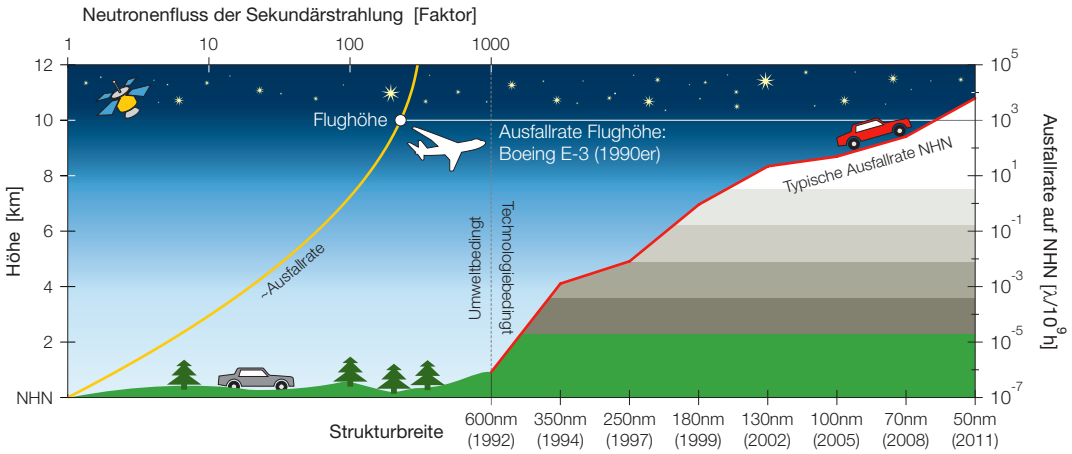


Abbildung 1.2: Illustrative Gegenüberstellung technologie- und umweltbedingter Zunahme transienter Hardwarefehler. Die Zahl der transienten Fehler nimmt erfahrungsgemäß proportional zur Neutronenflussdichte mit der Höhe zu. Entsprechende Schutzmaßnahmen sind daher, zum Beispiel in Flugzeugen, gängige Praxis. Die Ausfallrate steigt ferner technologiebedingt durch die Verringerung der Strukturbreiten. Sie bewegt sich mittlerweile in Regionen, die vergleichbar mit der Ausfallrate von Flugzeugen ist. Durch diese Entwicklung kann die Anfälligkeit moderner Hardware gegenüber transienten Fehlern in vielen Anwendungsfeldern nicht mehr vernachlässigt werden. Hier sind insbesondere Konsumgüter wie Fahrzeuge oder Medizingeräte zu nennen.

zehn Kilometern beispielsweise ist die Flussdichte um den Faktor 300 höher als auf *Normalhöhennull (NHN)*. Die in Flugzeugen eingesetzten Rechensysteme sind daher üblicherweise gegenüber transienten Fehlern abgesichert, zum Beispiel durch eine redundante Auslegung [Yeh96].

Zusätzlich zu den umweltbedingten Einflüssen unterliegt die Ausfallrate durch transiente Fehler einer technologiebedingten Zunahme. Diese Entwicklung lässt sich sehr anschaulich am Beispiel der ionisierenden Strahlung erläutern. Die Fehlerursache ist hier eine Ladungsverschiebung innerhalb des Halbleitermaterials – die Wirkungsweise ionisierender Partikel erläutert der Exkurs auf Seite 4. Damit sich ein Fehler tatsächlich manifestieren kann, muss die eingebrachte Ladung die zuvor bereits genannte kritische Ladung (Q_{crit}) übersteigen. Diese ergibt sich aus der Kapazität einer Speicherzelle und deren Versorgungsspannung und liegt bei gängigen Fertigungstechnologien im Bereich weniger Femtocoulomb (fC). Im Zuge der voranschreitenden Miniaturisierung sinken typischerweise sowohl die Versorgungsspannung als auch die Kapazität, wodurch immer weniger Elektronen für die Darstellung der Schaltungszustände zur Verfügung stehen. Dementsprechend reicht auch eine immer kleinere kinetische Teilchenenergie aus, um tatsächlich Fehler zu verursachen (vergleiche Exkurs, Seite 4). Darüber hinaus konnten Karnik und Hazucha [KH04] eine überproportionale Zunahme der Fehlerrate bei Senkung der Versorgungsspannung nachweisen.

Die Folgen dieser technologischen Entwicklung sind illustrativ in Abbildung 1.2 (rechts) dargestellt. Die Reduktion der Strukturbreiten von 600 nm auf 50 nm gehen nach Shivakumar et al. [SKK+02] mit einer geschätzten Zunahme der Ausfallrate um den Faktor 10^{10} einher. Diese Steigerung führt dazu, dass die Ausfallraten am Boden technologiebedingt inzwischen in die Region von 10^3 Fehlern je 10^9 Betriebsstunden vorstoßen. Zum Vergleich: Die von Taber und Normand [TN93] gemessene Ausfallrate des Speichersubsystems einer Boeing E-3 im Flug bewegt sich in derselben Größenordnung. Damit ist die Anfälligkeit moderner Hardware gegenüber transienten Fehlern zwischenzeitlich auf einem Niveau angekommen, auf dem sie in vielen Anwendungsfeldern nicht mehr vernachlässigt werden kann. Gleichzeitig werden heute immer mehr sicherheitskritische Funktionen in alltäglichen Konsumgütern mittels Software realisiert. Ein anschauliches Beispiel hierfür ist die zunehmende Zahl an Fahrerassistenzsystemen in modernen Fahrzeugen. Unabhän-

gig von diesem Beispiel ist eine genaue Vorhersage der zu erwartenden Fehlerraten weder für existierende noch für zukünftige Systeme trivial. Auch die Auswirkungen der ionisierenden Strahlung werden in der Literatur [ITY+10, DHW09, SSK+06, Con03] kontrovers diskutiert, da zum Beispiel mit der Verkleinerung der Strukturbreiten nicht nur die kritische Ladung sinkt, sondern auch die Fläche der Strukturelemente und damit die Wahrscheinlichkeit eines Treffers abnimmt. Zudem unterscheiden sich die Fehlerraten zwischen Speicher- und Logikelementen signifikant.

Hardware: Mehr Leistung und Parallelität, weniger Zuverlässigkeit

Insgesamt gehen Hersteller wie Intel [Bor05] davon aus, dass die Leistungsfähigkeit und Parallelität der Hardware in Zukunft weiter ansteigen und im Gegenzug die Zuverlässigkeit, unabhängig von den einzelnen Fehlerursachen, in Summe abnehmen wird. Diese Annahme wird durch aktuelle Feldstudien gestützt [DHW09]. So konnten beispielsweise Nightingale et al. [NDO11] eine mit der Technologiegeneration ansteigende Fehlerrate bei Desktopsystemen nachweisen.

1.2 Aufbau zuverlässiger Systeme aus unzuverlässiger Hardware

Die Auswirkungen transienter Fehler auf die Software sind üblicherweise schwerwiegend, da die Ausführung diese Art von Fehlfunktion weder erwartet noch überprüft. Die notwendige Fehlererkennung beziehungsweise Fehlertoleranz erfordert *Redundanz* und impliziert damit das Hinzufügen von Informationen, Ressourcen oder (Ausführungs-)Zeit. Ein bekanntes Beispiel für einen fehlertoleranten Systemaufbau ist die von Von Neumann [VN56] vorgeschlagene *dreifache Redundanz (TMR)* (engl. *triple modular redundancy*). Das Konzept des *Redundanzbereichs* beschreibt dabei den Wirkungsbereich der entsprechenden Maßnahme.

Fehlertoleranz in Hardware

Ein naheliegender Ansatz ist es, transiente Fehler an der Quelle zu bekämpfen und geeignete Maßnahmen auf Ebene der Hardware zu ergreifen.

Eine Möglichkeit liegt in der *Härtung der Schaltung* gegenüber ionisierenden Partikeln durch eine Erhöhung[4] der kritischen Ladung mittels zusätzlicher Kapazitäten oder durch zusätzliche Schaltungselemente, die trotz Ladungsverschiebung eine korrekte Funktion der Schaltung gewährleisten. Der Redundanzbereich erstreckt sich über alle gehärteten Teile der Schaltung und kann einzelne Funktionseinheiten oder auch den gesamten Prozessor umfassen. Schaltungshärtung wird aufgrund ihrer erheblichen Nachteile, wie größere Schaltungslatenzen und Kosten, üblicherweise für spezielle Anwendungszwecke, beispielsweise Luft- und Raumfahrt, eingesetzt. [CNV96, KVV+02]

Eine weitverbreitete Technik zum Schutz vor transienten Fehlern ist die *Codierung* (*ECC*) (engl. *error-correcting code*) der Daten. Hierfür werden zusätzliche, redundante Informationen zu den Nutzdaten hinzugefügt, welche die Erkennung oder sogar die Korrektur von Fehlern ermöglichen. Bekannte Beispiele sind Paritätsbits [Ham50] oder zyklische Redundanzprüfung (*CRC*) (engl. *cyclic redundancy check*) [Sha49]. Der Redundanzbereich ist dementsprechend auch auf Speichereinheiten (den Datenfluss) des Systems beschränkt. Die Anwendung auf Ausführungseinheiten und den Kontrollfluss ist im Allgemeinen nicht möglich. Hier kommen andere Verfahren wie

[4] Es liegt im Auge des Betrachters, ob dies als systematische Verbesserung oder Redundanz gesehen wird.

etwa arithmetische Codes (engl. *Residue Code* oder *AN Code*) oder Paritätsvorhersage auf Schaltungsebene (engl. *parity prediction circuits*) zum Einsatz, die jedoch aufgrund ihrer Komplexität weit weniger verbreitet sind [Nic03, For89].

Die umfassendste Fehlerabdeckung bietet jedoch die physikalische *Replikation* der Hardware. Die Fehlererkennung erfolgt hier über die redundante Ausführung und den Vergleich der Ausgangssignale. Entsprechend umfasst der Redundanzbereich das gesamte replizierte System. Neben Anzahl und Umfang der Replikate wird technisch zwischen synchroner (engl. *lockstepping*) und asynchroner Replikation (RMT) (engl. *redundant multithreading*) unterschieden. Bei der synchronen Replikation erfolgt die Ausführung der Replikate im Gleichtakt, üblicherweise verteilt auf (zwei) identische Prozessorkerne. Der Replikatzustand ist daher zu jedem Zeitpunkt identisch und kann fortlaufend verglichen werden, eine semantische Unterscheidung der Ausgangssignale ist daher nicht notwendig. Die erforderliche, taktgenaue Synchronisation ist allerdings sehr aufwendig. Demgegenüber ist die Ausführung bei der asynchronen Replikation nur lose gekoppelt, die Replikate werden hier unabhängig bis zu einem definierten Ausgabepunkt ausgeführt, an welchem die Ergebnisse (Signale) verglichen werden. Entsprechend ist die Implementierung zwar technisch einfacher, der Vergleich der Ausgabesignale ist jedoch anwendungsspezifisch und erfordert entsprechend zusätzliches, semantisches Wissen. Beide Ansätze sind in sicherheitskritischen Systemen weit verbreitet, beispielsweise in Form dreifach redundanter Avionik in modernen Flugzeugen. [SS98]

Maßnahmen auf der Ebene der Hardware haben die Vorteile, dass sie aus Sicht der Software keine Leistungseinbußen mit sich bringen und weitgehend transparent sind. Darüber hinaus ist der Redundanzbereich, abhängig von der gewählten Technik, sehr groß. Aus Sicht der Systemsoftware bedeutet dies, dass die nicht-funktionale Eigenschaft Zuverlässigkeit alleine durch die Hardware vorgegeben ist. Die Anforderungen an die Zuverlässigkeit und das konkrete Schutzbedürfnis hängen jedoch maßgeblich von der Anwendung ab. So ist zum Beispiel das primäre Ziel im Falle eines Netzwerkrouters lediglich die Erbringung einer gewissen Dienstgüte. Demgegenüber erfordert die Airbagsteuerung in einem Fahrzeug eine umfassende Fehlertoleranz. Letztlich sind die für den Anwendungsfall zulässigen Gesamtfehlerraten maßgebend. Hardwarelösungen bringen folglich auch eine ganze Reihe von Nachteilen mit sich. Zu diesen zählen vor allem die *hohen Kosten* und die *fehlende Selektivität*. So verdoppelt sich die notwendige Chipfläche zum Beispiel bei dem Einsatz der synchronen Replikation durch den notwendigen zweiten Prozessorkern, ohne dass die Leistungsfähigkeit des Gesamtsystems dadurch ansteigt. Dies gilt im Besonderen für fehlertolerante Systeme, die dreifach redundant ausgelegt sind. Hier können, abhängig von der Anwendung, weitere indirekte Kosten in Form von zusätzlichem Bauraum oder Gewicht hinzukommen. Darüber hinaus bestimmt die Anwendung mit den höchsten Anforderungen an die Ausfallrate die Auslegung des Gesamtsystems. In der Konsequenz wird auch die Ausführung aller anderen Anwendungen demselben hohen Schutzniveau unterworfen. Entsprechend sind anwendungsspezifische, maßgeschneiderte Lösungen in Hardware schwierig umzusetzen.

Fehlertoleranz in Software

Auf der anderen Seite lassen sich geeignete Maßnahmen zur Fehlererkennung und Fehlertoleranz auch auf der Ebene der Software umsetzen. Die grundlegenden Konzepte und Lösungen gleichen hier den Ansätzen der hardwarebasierten Fehlertoleranz und umfassen Codierung und Replikation von Daten- und Kontrollflüssen.

Typische Beispiele für Codierung sind Prüfsummen für Datenflüsse und signaturbasierte Kontrollflussüberwachung. Letztere realisiert die Fehlererkennung über einen Vergleich von vorausberechneten und in Abhängigkeit vom tatsächlichen Kontrollfluss errechneten Signaturen [GRRV06a]. Es existieren darüber hinaus auch kombinierte Verfahren, die unter Zuhilfenahme

von arithmetischen Codes sowohl den Kontroll- als auch Datenfluss absichern können [WF07]. Wie im Falle der Hardware ist die Komplexität und damit der Berechnungsaufwand verhältnismäßig hoch.

Eine bewährte Technik ist die softwarebasierte Redundanz in Form von Replikation [LABK90]. Diese kann in Software auf unterschiedlichen Ebenen implementiert werden. Eine Replikation durch *Voll-* [Gol73, VMw06] beziehungsweise *Paravirtualisierung* [BDGR97, BDF+03] (engl. *full- and paravirtualization*) zielt dabei auf die Nachbildung der hardwarebasierten Redundanz ab. Zu diesem Zweck wird die gesamte (virtuelle) Maschine repliziert und mehrfach auf der selben Hardware zur Ausführung gebracht. Es handelt sich hierbei um eine softwarebasierte Variante der asynchronen Hardwarereplikation. Da hier die gesamte Ausführung repliziert wird, ist der Redundanzbereich vergleichbar und schließt im Idealfall lediglich den Programmzähler aus. Entsprechend ist jedoch auch die Selektivität verhältnismäßig niedrig. Ein Beispiel für diesen Ansatz ist die Marathon EverRun Virtual Machine [TLM+09]. Ein anderes Extrem ist die Replikation einzelner Berechnungen, zum Beispiel ein mehrfaches Ausführen desselben Algorithmus durch die Anwendung. Hier beschränkt sich der Redundanzbereich auf die bloße Berechnung. Ein vielversprechender Mittelweg ist die Replikation von Prozessinkarnationen auf Ebene des Betriebssystems [SMR+07]. Hiermit lassen sich selektiv einzelne Anwendungsteile absichern. Der Redundanzbereich umfasst in diesem Fall die replizierte Ausführung der Prozesse.

Die Vor- beziehungsweise Nachteile der softwarebasierten Fehlertoleranz kehren sich im Vergleich zu den Maßnahmen auf Ebene der Hardware praktisch um. Zu den wesentlichen Nachteilen zählen der typischerweise höhere Ressourcenbedarf und der tendenziell kleinere Redundanzbereich [Muk08]. Letzteres ergibt sich aus Hardwarefunktion, die sich nicht in Software nachbilden beziehungsweise mit Redundanz versehen lassen. Der bereits genannte Programmzähler ist ein typisches Beispiel hierfür. Demgegenüber haben softwarebasierte Maßnahmen auch zahlreiche Vorteile. Hierzu zählt vor allem, dass sie selektiv und entsprechend des Schutzbedürfnisses der Anwendung eingebracht werden können. Durch die Unabhängigkeit von der Hardware ist zudem eine nachträgliche Absicherung beziehungsweise spätere Anpassung des Schutzniveaus möglich. Darüber hinaus wird der Entwicklungsprozess durch die Entkopplung von Hardware und Software potenziell vereinfacht [SMR+07].

Zuverlässigkeit durch softwarebasierte Redundanz

Mittels softwarebasierter Redundanz lässt sich die Zuverlässigkeit unabhängig von der Hardware beeinflussen. Ihre Leistungsfähigkeit und Parallelität kann so wahlweise zur Steigerung der Performanz oder zur Verbesserung der Zuverlässigkeit genutzt werden. Auch ermöglicht dies den Einsatz kostengünstiger Standardhardware in Bereichen, in denen dies bisher nicht möglich war beziehungsweise in Zukunft nicht mehr möglich sein wird. Problematisch sind jedoch Lücken im Redundanzbereich die sich in Form *kritischer Fehlerstellen* manifestieren und die generelle Anwendbarkeit softwarebasierter Redundanz gefährden.

Anwendungsbeispiel: Sicherheitskritische Regelungssysteme

Regelungssysteme sind heutzutage quasi omnipräsent und erfüllen die verschiedensten Aufgaben von der Steuerung großer Produktionsanlagen und Kraftwerken über Fahrerassistenzsysteme in Fahrzeugen bis hin zu Anwendungen im Bereich der Medizintechnik. Sie stellen damit mutmaßlich die Mehrheit der in Software realisierten sicherheitskritischen Funktionen und Anwendungen im Feld dar. Im Allgemeinen unterscheiden sie sich von anderen Anwendungen durch ihre Bindung an ein physikalisches System und die explizite Abhängigkeit von der Umwelt. Regelungssysteme

erfassen daher den Zustand der Umwelt mittels Sensoren, verarbeiten diese Daten und nutzen Aktuatoren zur gezielten Beeinflussung des physikalischen Systems. Entsprechend gehören sie zur Klasse der *Echtzeitsysteme*, da ihre Funktion neben der korrekten Berechnung maßgeblich von der rechtzeitigen Bereitstellung der Ergebnisse abhängt.

Sicherheitskritische Regelungssysteme stehen nicht nur durch ihre Anzahl exemplarisch für Anwendungen, die ein ausgewiesenes Schutzbedürfnis gegenüber transienten Fehlern aufweisen. Die zulässigen Fehlerraten sind in vielen Anwendungsfeldern durch entsprechende Sicherheitsnormen vorgegeben, wie zum Beispiel für Fahrzeuge durch die ISO 26262-5 [ISO11a]. Die Domäne der sicherheitskritischen Regelungssysteme steht daher im Fokus dieser Arbeit und dient im Folgenden als Anwendungsbeispiel.

1.3 Problembeschreibung und Forschungsfrage

Insgesamt nimmt die Zahl der in Software implementierten sicherheitskritischen Funktionen durch den technischen Fortschritt stetig zu und geltende Normen machen eine Berücksichtigung transienter Fehler in der Systemauslegung zunehmend unumgänglich [Bro06, EJ09]. Ein bewährter Ansatz um die notwendige Fehlererkennung bereitzustellen, ist der Einsatz der zuvor skizzierten Hardwareredundanz beziehungsweise speziell gehärteter Hardware. Diese Ansätze sind jedoch aufgrund der damit verbundenen hohen direkten wie indirekten Kosten und der fehlenden Selektivität, wie sie bei einer Mischung aus sicherheitskritischen und unkritischen Anwendungen (engl. *mixed-criticality systems*) erforderlich wäre, in vielen Bereichen nicht sinnvoll einsetzbar. Gerade bei Konsumgütern wie Fahrzeugen oder Medizingeräten herrscht ein enormer Kostendruck, der zusammen mit der zunehmenden Leistungsfähigkeit eingebetteter Systeme eben gerade dazu führt, dass immer mehr Funktionen auf einem Rechenknoten konsolidiert werden [Har06]. Das übergeordnete Ziel dieser Arbeit ist daher die selektive und systematische Beeinflussung der nicht-funktionalen Eigenschaft Zuverlässigkeit durch softwarebasierte Redundanz. Aus Sicht des Anwendungsentwicklers soll sich diese möglichst transparent und unabhängig von der Hardware einbringen lassen und gleichzeitig einen umfassenden Schutz vor transienten Fehlern bieten.

Lücken im Redundanzbereich

Softwarebasierte Redundanz kann selektiv angewendet werden und bietet gleichzeitig eine effektive Fehlererkennung im Rahmen einer replizierten Ausführung, zum Beispiel mittels der Replikation von Prozessinkarnationen. Das Problem an dieser Stelle sind die Lücken zwischen der hierdurch entstehenden Redundanzbereichen. Replikation auf Ebene der Software lässt *kritische Fehlerstellen* (*SPoF*) (engl. *single point of failure*) offen: Unreplizierte Bereiche, die nicht mehrfach aktiviert werden können und entsprechend ungeschützt auf der unzuverlässigen Hardware ausgeführt werden. Treten Fehler an diesen kritischen Stellen auf, ist ein Ausfall des Gesamtsystems trotz der umgebenden Redundanzmaßnahmen möglich. Zu den typischen Fehlerstellen eines dreifach redundanten Aufbaus zählen zum Beispiel die notwendige Aufteilung der Eingangssignale auf die Replikate, der Mehrheitsentscheid am Ende der replizierten Ausführung sowie die anschließende Verteilung der Ausgabesignale.

Obwohl diese Fehlerstellen üblicherweise sehr klein und kurz im Sinne der Ausführung sind, ist ihre Anzahl in hohem Maße anwendungsabhängig. Jeder Interaktionspunkt mit der Umgebung, bedingt zum Beispiel durch Datenaustausch mit anderen Anwendungen oder den Zugriff auf Systemdienste, führt zu einem Aufbruch der replizierten Ausführung. Weitere Problemstellen sind die Systemgrenzen. Eine Fehlererkennung und Absicherung der Sensorik oder Aktuatorik ist mittels replizierter Ausführung beispielsweise nicht möglich. Dies gilt allgemein für jede Art

von Übergang zwischen sicheren und unsicheren Domänen, wie zum Beispiel verteilte Systeme mit mehreren Rechenknoten. Darüber hinaus ist es zukünftig auch vorstellbar, dass Hersteller nur gewisse Teile der Hardware, beispielsweise einzelne Prozessorkerne, härten und solche Übergänge damit auch innerhalb eines Mehrkern-Rechensystems auftreten.

Die erzielbare Verbesserung der Zuverlässigkeit kann durch eine lückenhafte Implementierung der Redundanz drastisch reduziert werden [BCS69, Arn73]. Hieraus ergeben sich weiterreichende Auswirkungen auf die praktische Umsetzbarkeit softwarebasierter Redundanz: Die zuvor genannten Normen machen eine Bestimmung der Ausfallrate unumgänglich und erfordern daher üblicherweise eine Risikoanalyse der ungeschützten Teile und damit auch der kritischen Fehlerstellen. Da der Redundanzbereich von der konkreten Implementierung der Anwendung abhängt, ändert sich dieser potenziell mit jeder Modifikation der Anwendungssoftware. Darüber hinaus legen aktuelle Studien [NDO11] den Schluss nahe, dass die üblicherweise angenommene räumliche und zeitliche Gleichverteilung [Muk08] der Fehler in der Praxis nicht gegeben ist. Diese Eigenschaft erlaubt es normalerweise, die Fehlerwahrscheinlichkeit in Relation zur Ausführungszeit zu setzen. All dies macht die Risikoanalyse der kritischen Fehlerstellen in der Praxis äußerst schwierig und aufwendig und kann sich daher negativ auf die erzielbare Effektivität und den Entwicklungsaufwand softwarebasierter Fehlertoleranz auswirken.

Regelungsanwendungen replizieren

Eine weitere Herausforderung liegt in der Einbettung der Regelungsanwendung in die Umwelt und die Abhängigkeit vom Voranschreiten der Zeit, welche hohe Anforderungen an die Echtzeiteigenschaften des Systems stellt. Die nicht-funktionalen Eigenschaften Zuverlässigkeit und Rechtzeitigkeit stehen hierbei in einem Spannungsfeld, da sie sich gegenseitig beeinflussen können. Eine einfache, monolithische Replikation der gesamten Regelung ist daher in vielerlei Hinsicht unzweckmäßig. Auf der einen Seite ist dies durch die Kopplung an die Hardware in Form von Sensorik und Aktuatorik häufig nicht möglich. Auf der anderen Seite werden durch die Replikation die zeitlichen Eigenschaften des Systems deutlich verändert. Allem voran ist hier das Laufzeitverhalten zu nennen. Der klassische Regelungsentwurf stützt sich üblicherweise auf die Annahme, dass die Ausführung der Regelungsanwendung stets äquidistant und mit konstantem Laufzeitverhalten erfolgt [PBdST05, ÅW11]. Diese Annahmen sind im replizierten Fall verzerrt und unter Umständen nicht mehr vollständig erfüllt. Nicht zuletzt leidet die Replikation unter hohen Latenzen für die Fehlererkennung, da diese erst nach der mehrfachen Ausführung des gesamten Regelsystems durchgeführt werden kann. Daher ist vielmehr eine feingranulare Aufteilung der Regelungsanwendung notwendig, um einerseits die Ablaufplanung zu erleichtern und andererseits die Fehlererkennungslatenz auf ein angemessenes Niveau zu reduzieren. Darüber hinaus stellt sich die Frage, wie sich die hardwarespezifische Sensorik und Aktuatorik ebenfalls in den Redundanzbereich eingliedern lässt.

1.4 Lösungsansatz

Um die im vorangegangenen Abschnitt skizzierten Probleme zu lösen, ist ein zweiteiliger Ansatz nötig. Der im Rahmen dieser Arbeit entwickelte Lösungsansatz ist schematisch in Abbildung 1.3 dargestellt. Er ermöglicht die systematische und gezielte Beeinflussung der Zuverlässigkeitseigenschaften der Hardware im Allgemeinen und die Replikation sicherheitskritischer Regelungsanwendung unter der Berücksichtigung der zeitlichen Anforderungen und der regelungstechnischen Entwurfsverfahren im Speziellen. Von der Hardware kommend, bietet der Lösungsansatz zum einen das allgemeine technische *Rüstzeug* für die Eliminierung der verbliebenen kritischen Feh-

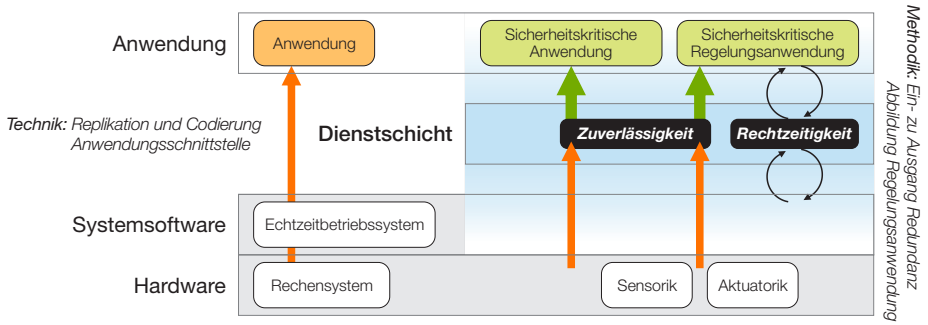


Abbildung 1.3: Schematische Einordnung des vorgeschlagenen Lösungsansatzes.
Der Ansatz fügt sich als Dienstschicht zwischen Anwendung und Systemsoftware (Echtzeitbetriebssystem) ein. Diese stellt *Techniken*, *Schnittellen* und *Methodiken* für die Eliminierung kritischer Fehlerstellen, den Aufbau eines geschlossenen Redundanzbereichs sowie für die feingranulare funktionale wie temporale Abbildung von Echtzeitregelungsanwendungen zur Verfügung.

lerstellen, um damit eine durchgehende softwarebasierte Fehlererkennung zu ermöglichen. Zum anderen stellt er aus Sicht des Anwendungsentwicklers geeignete *Methodiken* und *Schnittstellen* bereit, die es ermöglichen, sicherheitskritische Funktionen und Regelungsanwendungen selektiv und durchgängig abzusichern und gleichzeitig die sich ändernden zeitlichen Eigenschaften geeignet zwischen Anwendung und Systemsoftware auszutauschen. Der Ansatz gliedert sich dabei, wie in Abbildung 1.3 dargestellt, als Dienstschicht zwischen Anwendung und Systemsoftware ein.

Technische Maßnahmen zur Eliminierung kritischer Fehlerstellen

Ausgangspunkt ist die softwarebasierte Redundanz durch Replikation von Prozessinkarnationen auf Ebene des Betriebssystems. Das Ziel ist es, den Redundanzbereich auszudehnen und die kritischen Fehlerstellen in der softwarebasierten Redundanz zu eliminieren, ohne dabei die Selektivität des Grundansatzes zu verlieren. Der Ansatz unterscheidet dabei in Datenflüsse zwischen geschützten Domänen und die zwingend notwendigen *Mehrheitsentscheider* (engl. *voter*). Eine adäquate Lösung für die Absicherung der Datenflüsse ist der Einsatz fehlererkennender Codes. Die eigentliche Herausforderung liegt in den Mehrheitsentscheidern, da diese zusätzlich über einen eigenen Kontrollfluss verfügen. Für eine vollständige Fehlererkennung müssen diese daher um zusätzliche Maßnahmen zur Kontrollflussüberwachung erweitert werden. Das Ziel des Ansatzes ist es, eine geeignete *Kombination* aus *redundanter Ausführung* und *Codierungstechniken* in Daten- und Kontrollflüssen zu finden und ihre Eignung hinsichtlich einer durchgehenden Fehlererkennung gegenüber transienten Fehlern nachzuweisen.

Beitrag dieser Arbeit

Diese Arbeit eliminiert im Allgemeinen die kritischen Fehlerstellen in softwarebasierter Redundanz und ermöglicht so die systematische und vollständige Beeinflussung der Zuverlässigkeit unabhängig von der Hardware. Der Ansatz vereinfacht zudem im Speziellen die funktionale wie temporale Abbildung von Echtzeitregelungsanwendungen durch die Einführung einer anwendungsorientierten Schnittstelle für die Modularisierung, Replikation und Koordinierung der Regelungsaufgaben.

Methodik und Anwendungsschnittstelle

Die Replikation auf Betriebssystemebene bietet den Vorteil, dass sich viele Anwendungsfälle direkt darauf abbilden lassen [SMR+07]. Damit ist der Schutz vor transienten Fehlern für die Anwendung weitgehend transparent. Regelungssysteme stellen in diesem Punkt allerdings besondere Anforderungen. Sie sind, zumindest im Falle klassischer Regelungstechnik, konzeptbedingt abhängig von einer äquidistanten, konsekutiven und instantanen Ausführung der Anwendung.

Ziel des Ansatzes ist es daher, den Anwendungsentwickler bei der Aufteilung und Replikation der Regelungsaufgabe zu unterstützen und darüber hinaus zeitliche Abweichungen, die im Zuge der Ausführung entstehen, an die Anwendung zurückzumelden. Erreicht wird dies einerseits durch eine angepasste Architektur für Regelungssysteme, welche die Aufteilung der Regelungsaufgabe und deren Abbildung auf replizierbare Prozesse vereinfacht. Dies umfasst auch Maßnahmen zur Integration der Sensorik und Aktuatorik in den Redundanzbereich, um eine *Eingang-zu-Ausgang*-Fehlererkennung zu gewährleisten. Die in dem Ansatz erarbeitete *regelungstechnische Schnittstelle* zwischen Systemsoftware und Anwendung erlaubt eine Abstraktion von den temporalen Eigenschaften der Regelung und erleichtert so die Abbildung auf ein Echtzeitbetriebssystem. Weiterhin senkt die feingranulare Aufteilung der Regelungsanwendung die Latenz bis zu der Erkennung von Fehlern und ermöglicht darüber hinaus die individuelle Behandlung von besonders kritischen beziehungsweise unkritischen Pfaden in der Regelungsanwendung.

1.5 Kapitelüberblick

Diese Arbeit gliedert sich in zwei wesentliche Themenkomplexe: Zum einen in die Systemebene und die Eliminierung der kritischen Fehlerstellen in softwarebasierter Redundanz. Zum anderen die Anwendungsebene und die Umsetzung der Fehlertoleranz im Kontext von sicherheitskritischen Regelungsanwendungen.

Systemebene: Fehlertoleranzbausteine

Kapitel 2 führt zunächst in die Grundlagen softwarebasierter Fehlertoleranz ein und erläutert die für diese Arbeit relevanten Begrifflichkeiten und Techniken. Im weiteren Verlauf werden verwandte Arbeiten vorgestellt und diskutiert, welche sich für die Umsetzung der angestrebten Ziele und Anforderungen eignen.

Kapitel 3 charakterisiert zunächst die kritischen Fehlerstellen in der softwarebasierten Replikation. Darauf aufbeuend entwickelt dieses Kapitel den kombinierten Redundanzansatz CoRed (engl. *Combined Redundancy*) und damit das Rüstzeug für die gezielte Beeinflussung der Zuverlässigkeit auf der Systemebene. Hierzu zählt insbesondere der Entwurf des CoRed-Mehrheitsentscheiders und der zugehörigen Replikationsinfrastruktur.

Kapitel 4 zeigt die Effektivität des kombinierten Redundanzansatzes mittels systematischer Fehlerinjektion. Zu diesem Zweck werden zunächst die Grundlagen der Fehlerinjektion eingeführt. Darauf aufbauend erfolgt eine detaillierte Betrachtung der notwendigen Parametrierung der eingesetzten arithmetischen Codierungsverfahren. Schließlich wird die Implementierung von CoRed schrittweise bis auf die Maschineninstruktionsebene auf ihre Effektivität hin evaluiert.

Anwendungsebene: Replikation von Regelungsanwendungen

Kapitel 5 untersucht die Umsetzbarkeit des Gesamtansatzes anhand einer Fallstudie mit der Fluglageregelung eines Quadrocopters. Das System wird zu diesem Zweck entsprechend der CoRed Methodik von Grund auf redundant ausgelegt. Dies umfasst die gesamte Kette von der Sensorik über die Regelungsanwendung bis zur Aktuatorik. Die Evaluation erfolgt wiederum mittels systematischer Fehlerinjektion.

Kapitel 6 analysiert die Auswirkungen der Replikation auf die Regelgüte der Anwendung. Basierend auf dieser Analyse erweitert dieses Kapitel CoRed durch die Einführung einer geeigneten Schnittstelle zwischen Regelungsanwendung und Ausführungsumgebung sowie einer darauf aufbauenden Regelungsarchitektur für die Modularisierung der Anwendung zu CoRed@Control.

Zusammenfassung und Ausblick

Kapitel 7 fasst die wesentlichen Punkte dieser Arbeit zusammen und bewertet Sie entsprechend der eingangs formulierten Probleme und Ziele. Darüber hinaus gibt dieses Kapitel einen Ausblick auf weitergehende Forschungsfragen und zukünftige Arbeiten.

1.6 Typographische Konventionen

In der vorliegenden Arbeit werden verschiedene Schriftbildkonventionen zur Darstellung bestimmter Termini sowie zur Hervorhebung wichtiger Aussagen verwendet. Die Einführung neuer Begriffe erfolgt durch *Kursivschrift*. Diese dient darüber hinaus auch als allgemeines Stilmittel für die textbezogene Betonung oder Hervorhebung wichtiger Ergebnisse und Aussagen. Die Darstellung von Quellcode erfolgt in `nichtproportionaler` Schrift. Eigen- und Produktnamen sowie Werkzeuge werden – soweit nicht anderweitig vordefiniert – durch Kapitälchen gekennzeichnet.

Darüber hinaus werden an verschiedenen Stellen im Text Einschübe genutzt. Die jeweilige Bedeutung ergibt sich hierbei aus der Gestaltungsform:

Exzerpt

Fasst die wichtigsten Punkte des vorangegangenen Abschnitts zusammen.

Exkurs

Gibt über essentiellen Punkte hinausgehende Hintergrundinformationen.

1.7 Zugehörige Veröffentlichungen

Die in dieser Dissertation enthaltenen Ideen und Ergebnisse wurden zum Teil bereits in den folgenden Publikationen veröffentlicht:

Elsner, C., Ulbrich, P., Lohmann, D. und Schröder-Preikschat, W. (2010): Consistent Product Line Configuration Across File Type and Product Line Boundaries. In K. Kang (Hg.), Proceedings of the 14th Software Product Line Conference (SPLC '10), Band 6287 von *Lecture Notes in Computer Science*, Seiten 181–195. Springer-Verlag, Heidelberg, Germany. ISBN 978-3-642-15578-9. doi:10.1007/978-3-642-15579-6_13.

Schmid, R., Sobolev, S. und Ulbrich, P. (2010): Schutzrecht (Antrag) EP2442229-A1: High reliability method of data processing, and controller unit. (Eingereicht: 28. Juli 2010).

Ulbrich, P., Elsner, C., Hoffmann, M., Schmid, R. und Schröder-Preikschat, W. (2010): Using MARTE in Code-centric Real-time Projects Providing Evolution Support. In P. Boulet, D. Cancila, H. Espinoza und A. Morawiec (Hg.), Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED '10), Seiten 25–29.

Stilkerich, M., Schedel, J., Ulbrich, P., Schröder-Preikschat, W. und Lohmann, D. (2011): Escaping the Bonds of the Legacy: Step-Wise Migration to a Type-Safe Language in Safety-Critical Embedded Systems. In G. Karsai, A. Polze, D.H. Kim und W. Steiner (Hg.), Proceedings of the 14th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '11), Seiten 163–170. IEEE Computer Society Press. ISBN 978-0-7695-4368-0. doi:10.1109/ISORC.2011.29.

Ulbrich, P., Kapitza, R., Harkort, C., Schmid, R. und Schröder-Preikschat, W. (2011): I4Copter: An Adaptable and Modular Quadrotor Platform. In Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11), Seiten 380–396. ACM Press, New York, NY, USA. ISBN 978-1-4503-0113-8. (Zitiert auf den Seiten 105 und 107.)

Ulbrich, P., Franzmann, F., Harkort, C., Hoffmann, M., Klaus, T., Rebhan, A. und Schröder-Preikschat, W. (2012a): Taking Control: Modular and Adaptive Robotics Process Control Systems. In P. Payeur und P. Ben-Tzvi (Hg.), Proceedings of the 10th IEEE International Symposium on Robotic and Sensors Environments (ROSE '12), Seiten 55–60. IEEE Computer Society Press, Washington, DC, USA. ISBN 978-1-4673-2705-3. doi:10.1109/ROSE.2012.6402632. (Zitiert auf den Seiten 105, 106, 107 und 132.)

Ulbrich, P., Franzmann, F., Scheler, F. und Schröder-Preikschat, W. (2012b): Design by Uncertainty: Towards the Use of Measurement Uncertainty in Real-Time Systems. In Proceedings of the 7th International Symposium on Industrial Embedded Systems (SIES '12), Seiten 275–278. IEEE Computer Society Press, Washington, DC, USA. ISBN 978-1-4673-2685-8. doi:10.1109/SIES.2012.6356595.

Ulbrich, P., Hoffmann, M., Kapitza, R., Lohmann, D., Schröder-Preikschat, W. und Schmid, R. (2012c): Eliminating Single Points of Failure in Software-Based Redundancy. In Proceedings of the 9th European Dependable Computing Conference (EDCC '12), Seiten 49–60. IEEE Computer Society Press, Washington, DC, USA. ISBN 978-1-4673-0938-7. doi:10.1109/EDCC.2012.21. (Zitiert auf Seite 107.)

Hoffmann, M., Ulbrich, P., Dietrich, C., Schirmeier, H., Lohmann, D. und Schröder-Preikschat, W. (2014): A Practitioner's Guide to Software-based Soft-Error Mitigation Using AN-Codes. In Proceedings of the 15th IEEE International Symposium on High-Assurance Systems Engineering (HASE '14). IEEE Computer Society Press, Miami, Florida, USA. doi:10.1109/HASE.2014.14.

In (Ulbrich et al., 2010, 2011, 2012a, 2012c) habe ich den Hauptbeitrag zum Entwurf, der Implementierung sowie der Evaluation geleistet und war der Hauptautor der Publikation. In (Ulbrich et al., 2012b und Hoffmann et al., 2014) habe ich den Hauptbeitrag zum Entwurf, einen maßgeblichen Beitrag zu der Implementierung und einen signifikanten Beitrag zur Evaluation geleistet und war der Hauptautor der Publikation. In (Elsner et al., 2010 und Stillkerich et al., 2011) war ich Nebenautor der Publikation und habe einen maßgeblichen Betrag bei der Evaluation geleistet. In (Schmid, Sobolev und Ulbrich, 2010) habe ich den Hauptbeitrag zum Entwurf und der Implementierung geleistet und war Nebenautor des Patents.

Teil I

Systemebene

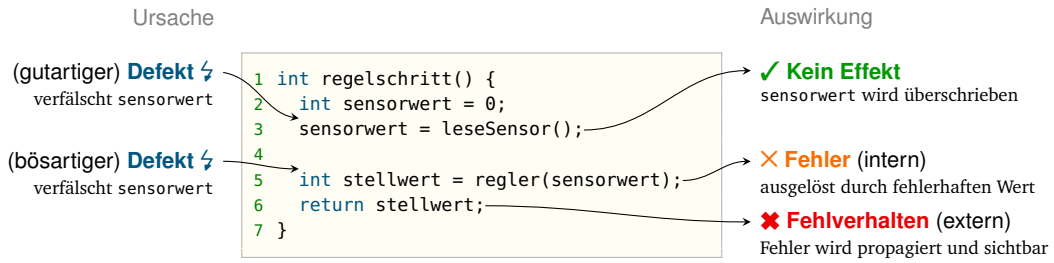
Fehlertoleranzbausteine

2
Softwarebasierte Fehlertoleranz

In der Einleitung wurden bereits die Ursachen sowie Auswirkungen transienter Fehler skizziert. Durch den technologischen Fortschritt nimmt zwar die Leistungsfähigkeit und Parallelität der Hardware stetig zu. Gleichzeitig ist jedoch davon auszugehen, dass ihre Zuverlässigkeit, unabhängig von den einzelnen Fehlerursachen, immer weiter abnehmen wird. Die Herausforderungen beim Entwurf sicherheitskritischer Anwendungen liegen einerseits im Aufbau zuverlässiger Systeme aus unzuverlässigen Komponenten. Die hierfür notwendige aktive Fehlererkennung beziehungsweise Fehlertoleranz erfordert in jedem Fall die Einbringung von Redundanz – sei es auf Ebene der Hardware oder der Software. Auf der anderen Seite führt gerade die Leistungsfähigkeit der Hardware zu immer neuen Anwendungsfeldern, wie auch zur Konsolidierung mehrerer Anwendungen mit unterschiedlichen Schutzbedürfnissen auf einem System.

Hardwarebasierte Fehlertoleranz ist in vielen Bereichen aufgrund der fehlenden Selektivität und der hohen direkten wie indirekten Kosten nicht zweckmäßig. Im Gegensatz dazu hat *softwarebasierte Fehlertoleranz* (engl. *software-implemented hardware fault-tolerance*) den Vorteil, dass sie selektiv und unabhängig von der Hardware angewendet und auch nachträglich in das System eingebracht werden kann. Im Rahmen dieser Arbeit wird sie zur gezielten Beeinflussung der Zuverlässigkeit auf Ebene der Systemsoftware genutzt. Problematisch sind in diesem Zusammenhang der eingeschränkte Redundanzbereich beziehungsweise die kritischen Fehlerstellen, wie zum Beispiel die Mehrheitsentscheider, in gängigen softwarebasierten Fehlertoleranzverfahren. Diese Einschränkungen haben mitunter weitreichende Auswirkungen auf die praktische Umsetzbarkeit der entsprechenden Ansätze (vgl. Kapitel 1.3). Der in dieser Arbeit verfolgte Lösungsansatz zielt daher darauf ab, diese kritischen Fehlerstellen zu eliminieren und eine möglichst lückenlose Abdeckung der zu schützenden Anwendungen zu erreichen.

Zu diesem Zweck führt dieses Kapitel zunächst in die Grundlagen (vgl. Kapitel 2.1) der softwarebasierten Fehlertoleranz ein und klärt die Frage, an welchen Stellen sich transiente Fehler manifestieren und wie sich diese auf die Strukturelemente der Software auswirken. Darauf aufbauend bestimmt die eingesetzte Redundanz, ob und in welchem Umfang Fehler erkannt und behandelt werden können. Die unterschiedlichen Fehlerfolgen stellen dabei einen wichtigen Indikator für kritische Fehlerstellen dar und führen letztlich zu einem entsprechenden Fehlermodell als Grundlage für die Bewertung der eingesetzten Fehlertoleranz. Darauf aufbauend folgt eine



Programmausschnitt 2.1: Fehlersymptomatik – Vom Defekt zum Fehlverhalten.
Transiente Hardwarefehler manifestieren sich auf Ebene der Software zunächst als *Defekt* (engl. *fault*). Dieser kann gutartig sein und durch den weiteren Ablauf verdeckt werden (oben). Andernfalls wirkt sich der Defekt auf den internen Zustand der Funktion `regelschritt()` aus (Mitte) und es kommt zur Auslösung eines *Fehlers* (engl. *error*). Wird dieser propagiert und nach außen sichtbar (unten), kommt es schließlich zum *Fehlverhalten* (engl. *failure*).

Gegenüberstellung grundlegender Entwurfsalternativen für die softwarebasierte Fehlertoleranz bestehend aus Redundanzart, Implementierungsebene, Fehlererkennung sowie Fehlerbehandlung.

Die Grundlagen dienen im Anschluss dazu, geeignete Fehlertoleranzmaßnahmen abzuleiten und die Abdeckung des Fehlermodells sowie den erzielbaren Redundanzbereich zu bestimmen. Schließlich wird der verfolgte Lösungsansatz ausgehend von den allgemeinen Ansätzen Replikation und Codierung in den Kontext verwandter Arbeiten (vgl. Kapitel 2.2) gesetzt.

2.1 Terminologie und Grundlagen

In Kapitel 1 wurden bereits einige Begrifflichkeiten wie *Fehler* oder *Redundanz* intuitiv eingeführt. Darauf aufbauend wurde der Begriff des *transienten Fehlers* als Synonym für transiente Hardwarefehler definiert, die sich als Bitfehler manifestieren und im Feld behandelt werden müssen. Der erste Teil dieses Kapitels diskutiert und erweitert die für diese Arbeit notwendigen Grundlagen im Hinblick auf die Domäne der softwarebasierten Fehlertoleranz. Zu diesem Zweck wird im Folgenden die logische Kette vom Auftreten eines Fehlers, über dessen Charakterisierung und Diagnose mittels Redundanz bis hin zu der Umsetzung eines fehlertoleranten Verhaltens näher betrachtet.

2.1.1 Fehler und deren Auswirkungen

Bisher wurde stets von transienten Fehlern gesprochen, jedoch führt nicht jeder Fehler auf Transistorebene automatisch zum Ausfall des gesamten Systems. Entsprechend wird daher, abhängig von der Ursache und der Sichtbarkeit, in weitere Fehlerklassen unterschieden, die vom Defekt bis zum Fehlverhalten reichen. Die sich daraus ergebende Fehlersymptomatik ist in Programmausschnitt 2.1 am Beispiel von zwei unabhängigen Einzelfehlern (Ursache, links) dargestellt.

Defekt, Fehler, Fehlverhalten – Fehlersymptomatik und Fehlerkette

Die eigentliche Ursache eines Fehlers ist ein *Defekt* (engl. *fault*) in den Strukturelementen der betrachteten Komponente. Ein solcher Defekt muss sich nicht zwangsläufig auf die Funktion der Komponente auswirken, weshalb zusätzlich *gutartige* (engl. *benign faults*) und *bösartige Defekte* (engl. *malicious faults*) unterschieden werden. Kommt es im Zuge des weiteren Betriebs zu einer Aktivierung eines bösartigen Defekts, führt dies tatsächlich zu einem *Fehler* (engl. *error*), der sich wiederum im internen Zustand der Komponente manifestiert. Bei einer Propagierung

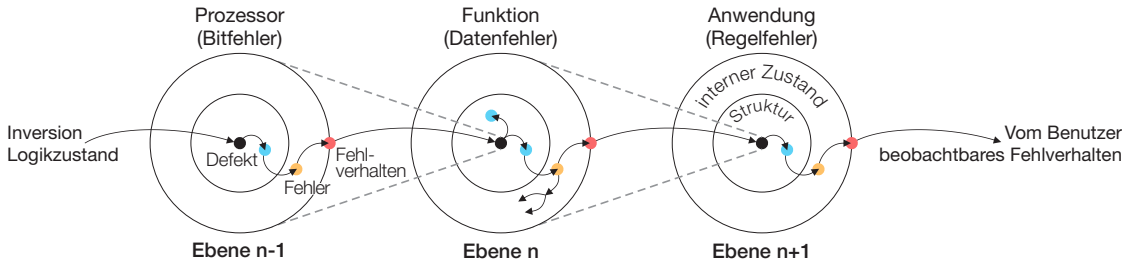


Abbildung 2.1: Fehlerausbreitung – Vom Bitfehler zum Systemversagen.
Die Betrachtungsebene bestimmt, ob es sich um einen Defekt oder ein Fehlverhalten handelt: So stellen Bitkipper ein Fehlverhalten der Hardware dar, sind jedoch aus Sicht der Software Defekte. Die Fehlerkette pflanzt sich dementsprechend über die Ebenen fort bis es potenziell zu einem beobachtbaren Fehlverhalten des Systems kommt. *Abbildung in Anlehnung an [Muk08].*

eines internen Fehlers bis an die Außengrenzen der Komponente tritt schließlich ein von außen beobachtbares *Fehlverhalten* (engl. *failure*) auf. [LABK90, ALRL04]

Diese Zusammenhänge lassen sich anschaulich anhand der Funktion `regelschritt()` in Programmausschnitt 2.1 erläutern. Die Strukturelemente sind in diesem Beispiel die Daten beziehungsweise Rechenanweisungen der Funktion. In ihnen manifestieren sich transiente Fehler zunächst als Defekt. Ein solcher mutiert im ersten Beispiel den Wert der internen Variablen `sensorwert`, kurz bevor diese in Zeile 3 mit einem aktuellen Wert überschrieben wird. Entsprechend ist die Fehlerkette an dieser Stelle unterbrochen und es tritt kein erkennbarer Effekt auf. Der zweite Defekt verändert `sensorwert` hingegen nach dessen Aktualisierung und wird damit bei der weiteren Verwendung durch die Funktion `regler()` in Zeile 5 aktiviert. In der Folge berechnet `regler()` einen inkorrekten Stellwert und es kommt zu einem internen Fehler. Die Fehlerkette setzt sich durch die Rückgabe der fehlerhaften Variablen `stellwert` fort und wird schließlich zu einem von außen durch ihren Aufrufer beobachtbar Fehlverhalten der Funktion `regelschritt()`.

Fehlerfortpflanzung und Sichtbarkeit

Ab wann ein Fehler beobachtbar und damit zum Fehlverhalten wird, hängt dabei maßgeblich von der Sichtbarkeit beziehungsweise der Betrachtungsebene ab. Grundsätzlich erscheint ein Fehlverhalten auf der aktuellen Ebene wiederum als Defekt auf der darüber liegenden Ebene. Die Fehlerkette pflanzt sich entsprechend von Ebene zu Ebene fort, bis sie schließlich in einem vom Benutzer beobachtbaren Fehlverhalten mündet. Ob sich ein Bitfehler tatsächlich auswirkt, hängt dementsprechend davon ab, ob sich der Fehler durch die Ebenen fortpflanzen kann, ohne maskiert zu werden. [Muk08, ALRL04]

Abbildung 2.1 veranschaulicht die Fehlerfortpflanzung durch die Ebenen des Gesamtsystems: Initial führt ein transienter Fehler zu einer Umladung in einem Transistor und damit zu einer Inversion seines Zustands. Dies zeigt sich aus Sicht des Prozessors (Ebene n-1) zunächst als Defekt in Form eines Bitfehlers. Befindet sich dieser in einem im weiteren Verlauf der Ausführung genutzten Teil des Prozessors, zum Beispiel in einem Register, kommt es zur Fehlerfortpflanzung und damit schließlich zum Fehlverhalten des Prozessors. Bezogen auf die Funktion (Ebene n) zeigt sich der fehlerhafte Registerwert wiederum als Defekt, zum Beispiel als Datenfehler in der Variablen `sensorwert` (vgl. Programmausschnitt 2.1). Dieser kann abermals maskiert werden oder bis hin zum Fehlverhalten der Funktion und der Rückgabe eines fehlerhaften Stellwerts führen. Analog manifestiert sich das Fehlverhalten der Funktion wiederum als Defekt in den Strukturelementen der Regelungsanwendung (Ebene n+1), welcher sich abermals bis zu einem vom Benutzer beobachtbaren Fehlverhalten und dem Versagen der Regelung fortpflanzen kann.

Die initiale Definition (siehe Seite 3) des Begriffs *transienter Fehler* lässt sich damit auf die Klasse der transienten Hardwarefehler verfeinern, die zu einem Fehlverhalten der Hardware und zu bösartigen Defekten in den Strukturelementen der Software führen.

Fehlertoleranzmaßnahmen: Erweiterte Fehlerklassifizierung

Transiente Fehler werden maskiert und bleiben folgenlos, falls sie sich auf einer Ebene lediglich als gutartige Defekte manifestieren. Gezielt lässt sich die Fehlerkette dagegen nur durch den Einsatz von entsprechenden Fehlertoleranzmaßnahmen unterbrechen. Um ihre Effektivität klassifizieren zu können, ist die bisherige Einteilung der Fehlerfolgen in Maskierung oder Fehlverhalten nicht mehr ausreichend. Letzteres wird daher, insbesondere in der Gegenwart von Fehlertoleranz, häufig zusätzlich in die folgenden zwei Fehlerklassen aufgeteilt [MER05, Muk08]:

Unerkannte Datenfehler (SDC) (engl. *silent data corruptions*) führen in jedem Fall zu einem Fehlverhalten. Sie stellen die schwerwiegendste Folge eines transienten Fehlers dar, da das System in Konsequenz unbemerkt fehlerhafte Ergebnisse liefert. Folglich haben an dieser Stelle auch alle eventuell vorhandenen Mechanismen zur Fehlerkennung versagt. Erschwerend kommt hinzu, dass häufig keine direkte Korrelation zwischen einem SDC und dem beobachtbaren Fehlverhalten beziehungsweise dem Ausfall des Systems besteht.

Erkannte nicht behebbare Fehler (DUE) (engl. *detected unrecoverable errors*) konnten erfolgreich detektiert werden, eine Maskierung des Fehlers war jedoch nicht möglich. Typischerweise werden DUEs als weniger schwerwiegend eingestuft, da sie eine gefährliche Korruption der Ergebnisse vermeiden und unmittelbar zu einem erkennbaren Fehlverhalten führen. Abhängig davon, ob der Fehler auf einer höheren Ebene tolerierbar ist, führt dies in der Folge zur Einstellung der Funktion (engl. *fail stop*) oder zu einem Neustart (engl. *recovery*) und damit einer zeitweisen Nichtverfügbarkeit des Systems. Im Falle von Fehlererkennung entsprechen DUEs dem erwarteten Verhalten. Darüber hinaus sind sie jedoch ein Indiz für fehlende Redundanz bei der Fehlerkorrektur. Ein anschauliches Beispiel hierfür sind fehlerkorrigierende Codes, welche üblicherweise mehr Bitfehler erkennen als korrigieren können.

Insbesondere die SDC-Fehlerrate ist daher in der Praxis ein wichtiges Maß für die Zuverlässigkeit des Systems beziehungsweise die Effektivität der eingesetzten Fehlertoleranzmaßnahmen – Hersteller geben die Fehlerraten daher häufig gesondert nach SDC und DUE an.

> **Unerkannte Datenfehler – Kritische Fehlerstellen**
>
> *Unerkannte Datenfehler* (SDC) treten bei fehlender beziehungsweise unzureichender Redundanz auf und stellen damit einen wichtigen Indikator für kritische Fehlerstellen in Fehlertoleranzmaßnahmen dar. Entsprechend besitzen sie eine besondere Relevanz bei der Bewertung von deren Effektivität.

Fehlermodell: Symptomatische Auswirkungen transienter Fehler auf der Softwareebene

Ob und in welchem Umfang softwarebasierte Fehlertoleranz unerkannte Datenfehler zulässt, hängt von der durch sie bereitgestellten Fehlerabdeckung ab. Die Fehlererkennung basiert dabei stets auf den Strukturelementen der Implementierungsebene und den dort erkennbaren symptomatischen Auswirkungen transienter Fehler. Die elementarsten Strukturelemente der Software stellen die Anweisungen der Maschinensprache dar. Daher treten die Symptome transienter Fehler, aus

Sicht der Software, initial auf der Instruktionsebene zutage. Diese lassen sich, unabhängig von der konkreten Ausprägung der Hardware, in drei grundlegende Symptome einteilen: *Operator-*, *Operanden-* und *Berechnungsfehler*. [For89]

```
ld  a = [mem_a], b = [mem_b]
add c = a, b
st  [mem_c] = c
```

Fehlerfreier Code

```
ld  a = [mem_a], b = [mem_b]
add c = a, b // c != a+b
st  [mem_c] = c
```

Berechnungsfehler

```
ld  a = [mem_a], b = [mem_b]
mult c = a, b // * statt +
st  [mem_c] = c
```

Operatorfehler

```
ld  a = [mem_x], b = [mem_b]
add c = a, b // a fehlerhaft
st  [mem_y] = c
```

Operandenfehler

Programmausschnitt 2.2: Symptomatische Auswirkungen transienter Fehler – Fehlermodell auf Instruktionsebene. Mögliche Auswirkungen transienter Fehler auf Instruktionsebene am Beispiel einer einfachen Addition. Die hervorgehobenen Stellen markieren exemplarisch die Symptome, die durch die Mutation der Operanden beziehungsweise Operatoren in der jeweiligen Fehlerklasse entstehen.

Die Fehlersymptomatik lässt sich anschaulich anhand der in Programmausschnitt 2.2 gezeigten Beispiele erläutern. Die Auswirkungen sind im Einzelnen:

Berechnungsfehler führen, obwohl sowohl Operator als auch Operanden korrekt sind, zu einem falschen beziehungsweise unerwarteten Ergebnis einer Operation. Im konkreten Beispiel in Programmausschnitt 2.2 wird die Addition von `a` und `b` plangemäß durchgeführt, das Ergebnis `c` ist jedoch fehlerhaft. Die Ursache kann beispielsweise in einem Fehlverhalten der entsprechenden *Arithmetisch-logischen Einheit (ALU)* (engl. *arithmetic logic unit*) oder einer der Berechnung folgenden Verfälschung des Ergebnisses liegen.

Operatorfehler bezeichnen die Veränderung des Operators. Dadurch kann zum Beispiel die Addition (`add`) zu einer Multiplikation (`mult`) verfälscht werden. Das Ergebnis `c` dieser Multiplikation ist an sich korrekt, entspricht jedoch nicht der erwarteten Addition und ist daher im weiteren Verlauf unbrauchbar. Eine Ursache für diese Fehlersymptomatik kann in der Verfälschung der entsprechenden Maschinenanweisung durch einen Bitfehler liegen. Ebenso kann ein transienter Fehler zu einer Änderung der Instruktionsreihenfolge, und damit implizit zu einem falschen beziehungsweise unerwarteten Operator führen, zum Beispiel durch eine Verfälschung des Programmzählers.

Operandenfehler haben, je nach Ausprägung, unterschiedliche Auswirkungen. Ein offensichtlicher Operandenfehler liegt bei einer Verfälschung des eigentlichen Datums vor, zum Beispiel durch einen Bitfehler in dem von `a` belegten Prozessorregister. Dieser *Datenfehler* (engl. *data error*) führt aufgrund der fehlerhaften Eingaben wiederum zu einem fehlerhaften Ergebnis `c` der Addition. Dieselbe Symptomatik ergibt sich durch einen Austausch des eigentlichen Operanden durch einen *Adressfehler* (engl. *address error*) beim Laden des entsprechenden Datums. In Programmausschnitt 2.2 wird beispielsweise die Adresse von `a` korrumpiert (`mem_x`) und verweist anschließend auf `x`. Folglich wird der korrekte Wert von `x` geladen und entspricht nun vermeintlich `a`. Wiederum entsprechen die Eingaben der Addition nicht den Erwartungen und es kommt zu einem fehlerhaften Ergebnis. Schließlich können Operandenfehler auch zu einem Verlust eines Datums durch einen *Aktualisierungsfehler* (engl. *lost update*) führen, falls sich der Adressfehler bei einem schreibenden Zugriff auswirkt. Im konkreten Beispiel wird `c` fälschlicherweise an die Speicheradresse `mem_y` geschrieben. In der Folge weist der Operand `c` einen veralteten Wert auf.

Komplexere Fehlermuster lassen sich grundsätzlich durch Kombination der genannten Symptome, zum Beispiel als Folge von Operator- und Operandenfehlern, nachbilden.

Die diesem Fehlermodell zugrunde liegende Annahme ist die immanente Beeinflussung der Programmausführung durch bösartige Defekte. Durch die Fehlerfortpflanzung ist es dabei unerheblich, ob ein transienter Fehler direkt auf den Prozessor wirkt und dort einen internen Defekt auslöst, oder ob es sich um ein Fehlverhalten einer externen Komponente handelt, zum Beispiel dem Speichersubsystem. Entsprechend kann das gesamte Fehlermodell auch in andere Betrachtungsebenen überführt werden. So werden beispielsweise Fehler auf der Ebene der Hochsprache C in Zuweisungs-, Prüf-, Schnittstellen- und algorithmische Fehler klassifiziert [DM02]. Einzelfehler in den Strukturelementen der Hochsprache lassen sich hierbei auf Fehler beziehungsweise Fehlerfolgen der Instruktionsebene abbilden. Im Allgemeinen lassen sich Fehlermodelle unterschiedlicher Ebenen ineinander überführen [GRRV06a].

Das Fehlermodell auf Instruktionsebene eignet sich aufgrund seiner Unabhängigkeit von der eingesetzten Hochsprache und der überschaubaren Zahl an Fehlerklassen besonders für die weitere Betrachtung verwandter Arbeiten und findet daher im Folgenden Anwendung.

Fehlermodell – Auswirkungen transienter Fehler

Unabhängig von der darunter liegenden Hardware lassen sich die symptomatischen Auswirkungen transienter Fehler auf Instruktionsebene in *Operator-*, *Operanden-* und *Berechnungsfehler* einteilen [For89]. Das Fehlermodell ist daher ein zentraler Aspekt für die Bewertung softwarebasierter Fehlertoleranz im Allgemeinen und für den Entwurf des angestrebten Lösungsansatzes in Kapitel 3 im Speziellen.

2.1.2 Redundanz als Mittel zur Fehlertoleranz

Die Grundvoraussetzung für die Erkennung und gegebenenfalls die Behandlung von Fehlern, beziehungsweise deren Folgen, ist *Redundanz* (engl. *redundancy*). Diese bezeichnet gemäß DIN 40041 [DIN90] zusätzliche Funktionalität, die nicht für die Erfüllung der eigentlichen Aufgabe des Systems notwendig ist. Entsprechend ist Redundanz, und damit auch jede Art der Fehlertoleranz, inhärent mit zusätzlichen Kosten beziehungsweise zusätzlicher Komplexität behaftet – unabhängig von ihrer Art, Implementierung oder der Ebene, auf der sie eingesetzt wird. [Pra96]

Redundanzarten

Im Verlauf von Kapitel 1.2 wurde Redundanz gemäß ihrer Umsetzung in Hardware beziehungsweise Software klassifiziert und konkrete Redundanztechniken wie Replikation und Codierung eingeführt. Unabhängig davon werden im Allgemeinen vier *Redundanzarten* unterschieden. Diese sind Redundanz in Information, Zeit, Funktion und Struktur [Ech90]:

Informationsredundanz (engl. *information redundancy*) umfasst im weitesten Sinne jede Art von zusätzlicher Information, die neben den eigentlichen Nutzdaten vorhanden ist und für die Fehlererkennung herangezogen werden kann. Ein einfaches Beispiel sind getrennt geführte Brutto- und Nettobeträge, die sich mit Kenntnis des Mehrwertsteuersatzes ineinander überführen lassen. Im Allgemeinen erfolgt der strukturierte Einsatz von Informationsredundanz durch die Codierung der Information (Datum) gemäß spezieller Codierungsvorschriften, welche auf eine möglichst hohe Fehlererkennung hin ausgelegt sind. Das entstehende Codewort verliert seine Gültigkeit, sobald es den Richtlinien nicht mehr genügt. Umgekehrt

wird die ursprüngliche Information durch Decodieren wiederhergestellt und die Redundanz entfernt. Bekannte Vertreter hierfür sind die bereits genannten Paritätsbits oder die zyklische Redundanzprüfung (CRC).

Funktionelle Redundanz (engl. *functional redundancy*) bezeichnet die Erweiterung des Systems um zusätzliche Funktionen, die für den regulären Betrieb entbehrlich sind und sich in ihrer Umsetzung unterscheiden. Dies umfasst zum Beispiel den Einsatz von *Funktionswächtern* (engl. *watchdogs*), welche bei der Überschreitung von definierten Schwellenwerten entsprechende Maßnahmen einleiten. Ebenso zählen hierzu Funktionen, welche den gleichen Sachverhalt auf unterschiedliche Weise herleiten. Ein Beispiel ist die Bestimmung einer Ventilstellung mithilfe eines Stellungsgebers beziehungsweise alternativ eines Durchflussmengenmessers. Die in Kapitel 1.2 genannte Härtung auf Schaltungsebene ist ebenfalls der funktionellen Redundanz zuzurechnen, da die zusätzlichen Schaltungselemente der Überwachung der normalen Funktion dienen.

Strukturelle Redundanz (engl. *structural redundancy*) beschreibt die Erweiterung des Systems um zusätzliche Komponenten, die für den Normalbetrieb nicht notwendig sind. Im Gegensatz zur funktionellen Redundanz sind die Komponenten hierbei gleichartig und erfüllen dieselbe Nutzfunktion. Strukturelle Redundanz führt dementsprechend zu *Replikation*, wobei es sich um einzelne Teile der Hardware beziehungsweise der Software oder das System als Ganzes handeln kann. Ein bekanntes Beispiel ist die redundante Auslegung sicherheitskritischer Komponenten in Flugzeugen [Yeh96]. Diese umfasst üblicherweise nicht nur die eigentlichen Rechensysteme, sondern vielmehr alle sicherheitsrelevanten Baugruppen von den Sensoren bis hin zu den Aktuatoren an den Steuerflächen.

Zeitredundanz (engl. *time redundancy*) bezieht sich schliesslich auf die Bereitstellung von zusätzlicher, über den Bedarf des Normalbetriebs hinausgehender, Zeit. Diese kann für die Fehlererkennung, Fehlerbehebung oder zur Wiederholung von Funktionen eingesetzt werden. Die überdimensionierte Auslegung eines periodischen Echtzeitsystems ist ein gängiges Beispiel für Zeitredundanz. Die aus der ungenutzten Rechenzeit entstehenden Puffer können im Fehlerfall entsprechend genutzt werden. Ebenfalls in die Klasse der Zeitredundanz fallen Algorithmen, deren Ergebnis sich mit der Anzahl der Wiederholungen verbessert.

In der Praxis kommen häufig Mischformen der genannten Redundanzarten vor. So erfordert die Umsetzung von Informationsredundanz mittels Codierung auch zusätzliche Zeit für deren Berechnung. Ebenso umfasst strukturelle Redundanz in Form von Replikation im Allgemeinen auch redundante Informationen und Zeit. Die Fehlererkennung basiert dabei jedoch zumeist auf einer der genannten Redundanzarten.

Koordinierter Einsatz: Redundanz – Strategie – Implementierungsebene

Generell ermöglicht erst der koordinierte Einsatz von Redundanz die Implementierung von Fehlertoleranz. Abbildung 2.2 zeigt exemplarisch den Weg von der grundlegenden Redundanz über eine Strategie zu deren Nutzung bis hin zu einer spezifischen Implementierung. Die Redundanzart gibt lediglich vor *womit* Fehler erkannt und damit letztlich Fehlertoleranz erreicht werden kann. Erst der Entwurf einer darauf aufbauenden *Redundanzstrategie* beschreibt das *Wie* und bestimmt damit maßgeblich in welchem Umfang Redundanz hinzugefügt und ob dadurch Fehlererkennung, Fehlerdiagnose, Fehlerbehandlung, Fehlerkorrektur oder Fehlermaskierung (vgl. Kapitel 2.1.3) ermöglicht wird. Es existieren beispielsweise verschiedene Strategien zur Replikation mit unterschiedlichen Eigenschaften. So lassen sich Einzelfehler durch *zweifache Redundanz (DMR)* (engl. *double modular redundancy*) im Allgemeinen lediglich erkennen, wohingegen durch den

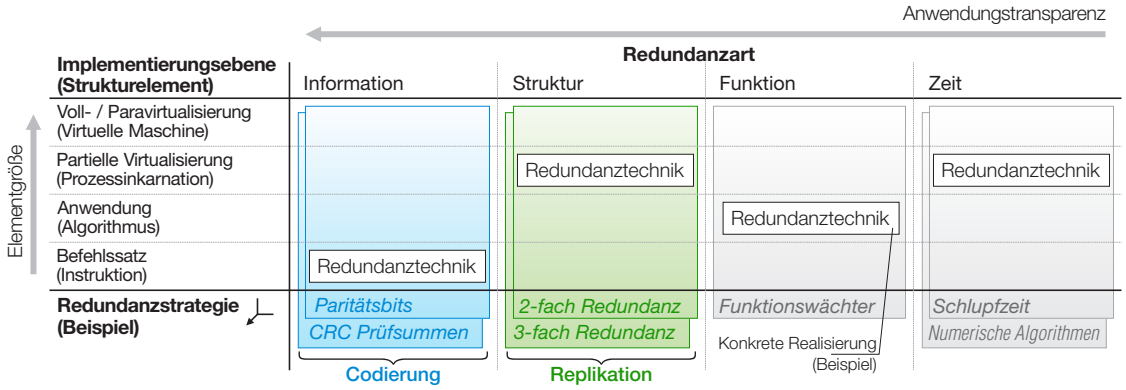


Abbildung 2.2: Entwurfsentscheidungen bei der Implementierung von Fehlertoleranz: Redundanz – Strategie – Technik. Fehlertoleranz ergibt sich aus dem koordinierten Einsatz der vier *Redundanzarten*. Die hierbei eingesetzten Vorgehensweisen, wie Replikation oder Codierung, lassen sich in *Redundanzstrategien* klassifizieren. Sie bestimmen die grundlegenden Eigenschaften wie Fehlererkennung, -diagnose oder -maskierung und geben ein Entwurfsmuster für die Implementierung vor, zum Beispiel die Anzahl der Replikate oder die Art der Codierungsvorschrift. Die Umsetzung in Form einer konkreten *Redundanztechnik* erfolgt schließlich auf einer *Implementierungsebene*. Diese Folge von Entwurfsentscheidungen bestimmt letztlich auf welche Strukturelemente Redundanz angewendet und wie dadurch Fehler erkannt werden können.

Einsatz eines weiteren Replikats und dreifacher Redundanz (TMR) zusätzlich Fehlerdiagnose (Lokalisierung) und Fehlermaskierung ermöglicht werden. Im Falle der Informationsredundanz und Codierung beschreibt die Strategie die abstrakte Codierungsvorschrift und bestimmt damit wiederum die resultierenden Fehlertoleranzeigenschaften. Zum Beispiel erlaubt die Codierung mittels einfacher Paritätsbits lediglich die Erkennung von Einzelbitfehlern. Demgegenüber ermöglichen beispielsweise mehrdimensionale Paritätsbits, sogenannte Hamming-Codes [Ham50], neben der Erkennung von Mehrfachfehlern auch die Fehlerkorrektur.

Die eigentliche Implementierung einer auf einer abstrakten Strategie basierenden *Redundanztechnik* (vgl. Abbildung 2.2) erfolgt schließlich auf einer bestimmten *Implementierungsebene* des Systems [Muk08], von der Befehlssatzebene bis hin zur Vollvirtualisierung des Systems. Die eingefügte Redundanz bezieht sich dabei, wie schon die Fehlerbetrachtung (vgl. Kapitel 2.1.1), stets auf die Strukturelemente der für die Umsetzung gewählten Ebene. Im Falle der Replikation lassen sich entsprechend Instruktionen, Berechnungen, Prozessinkarnationen oder ganze (virtuelle) Maschinen replizieren. Die Implementierungsebene hat zwar keinen Einfluss auf die grundlegenden Fehlertoleranzeigenschaften, bestimmt jedoch maßgeblich den Umfang und die Granularität der Komponenten, in denen Fehlertoleranz möglich ist. Die Klassifizierung in Redundanztechnik, Redundanzstrategie und Redundanzart erleichtert somit die Bewertung der verschiedenen Techniken und Ansätze zur Behandlung von transienten Fehlern.

Der Redundanzbereich

Unerkannte Datenfehler haben ihren Ursprung in den bereits erwähnten Lücken gängiger Fehlertoleranzmaßnahmen. Für ein besseres Verständnis ihrer Funktionsweise ist nicht nur relevant wie und womit Fehlertoleranz erreicht wird. Vielmehr ist auch entscheidend wann und wo Redundanz zur Verfügung steht. Ein anschauliches Beispiel für diese Problematik ist die in Abbildung 2.3 dargestellte dreifache Redundanz. Die redundante Ausführung erlaubt die uneingeschränkte Fehlererkennung innerhalb der replizierten Komponenten. Hierfür müssen allerdings die notwendigen Eingaben vervielfältigt und auf die Replikate verteilt werden. Ebenso ist ein Vergleich der Ausgaben am Ende der Ausführung unerlässlich. In diesen Teilen ist jedoch keine Fehlererkennung

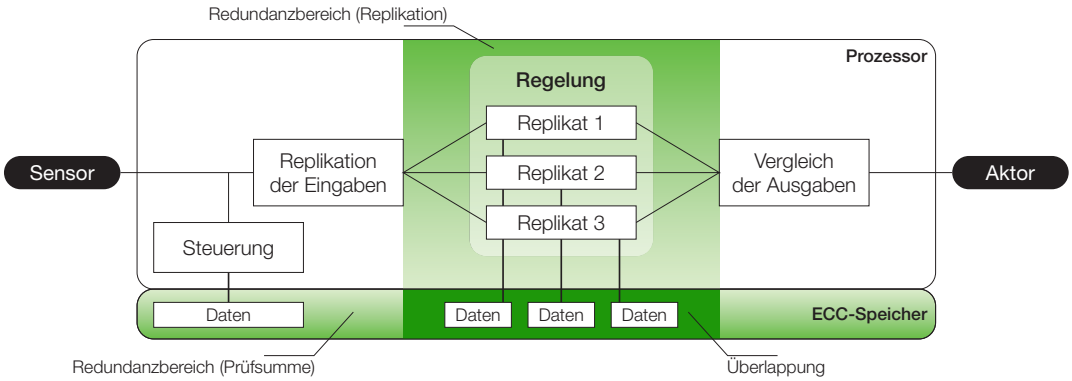


Abbildung 2.3: Die Grenzen der Fehlertoleranz: Der Redundanzbereich.
Die Eigenschaften der Redundanzstrategie charakterisieren die Effektivität einer Fehlertoleranzmaßnahme. Der *Redundanzbereich* beschreibt hingegen den räumlichen und zeitlichen Wirkungsbereich der entsprechenden Maßnahme. Dieser beschränkt sich im Falle der replizierten Regelung räumlich auf die eigentlichen Replikate. Die Replikation der Eingaben und der Vergleich der Ausgaben sind zwar Teil der Fehlertoleranzmaßnahme, liegen jedoch außerhalb des farblich hervorgehobenen Redundanzbereichs. Im Gegensatz dazu erstreckt sich der Redundanzbereich des ECC-Speichers räumlich über alle Elemente, umfasst jedoch zeitlich nur die Verweildauer im Speicher.

möglich. Reinhard und Mukherjee [RM00] nutzen das Konzept des *Replikationsbereichs (SoR)* (engl. *sphere of replication*), um diesen Zusammenhang zu beschreiben. Fehler, die innerhalb des replizierten Bereichs auftreten und sich bis zu dessen Grenzen propagieren, werden durch die Vergleichsoperation erkannt.

In ähnlicher Weise lassen sich auch andere Fehlertoleranzmaßnahmen charakterisieren. So beschränkt sich die Fehlererkennung durch Prüfsummen auf den Bereich von der Codierung bis zur Decodierung der Werte. Der entstehende Wirkungsbereich hängt daher auch von der Implementierungsebene ab und ist eine Eigenschaft, die maßgeblich von der konkreten Redundanztechnik beeinflusst wird. Im Falle von ECC-Speicher, wie im Beispiel in Abbildung 2.3 dargestellt, ist die Redundanz und somit die Fähigkeit zur Fehlererkennung auf die Verweildauer im Speicher limitiert. Da die Codierung beziehungsweise Decodierung im Zuge des Speicherzugriffs direkt im Speicherbaustein erfolgt, ist die weitere Verwendung der Daten im Prozessor im Rahmen der ECC-Codierung folglich nicht abgedeckt.

Im Rahmen dieser Arbeit dient daher das generalisierte Konzept des *Redundanzbereichs* der örtlichen und zeitlichen Charakterisierung der eingesetzten Fehlertoleranz. Ausgehend von der Effektivität der gewählten Redundanzstrategie ist eine Fehlererkennung innerhalb dieser Bereiche beziehungsweise an deren Grenzen stets gegeben. Abbildung 2.3 zeigt die sich ergebenden Redundanzbereiche farblich hervorgehoben. In diesem Beispiel ergeben sich zwei Einzelbereiche aus der eingesetzten Replikation beziehungsweise Codierung. Diese ergänzen sich zu einem zusammenhängenden globalen Redundanzbereich, wobei sich die Maßnahmen in den replizierten Daten überlappen – an dieser Stelle ist die Fehlertoleranz selbst redundant.

2.1.3 Fehlertolerantes Verhalten

Für die Umsetzung von Fehlertoleranz reicht der strategische Einsatz von Redundanz alleine noch nicht aus. Erst wenn sich Fehler in den redundanten Strukturelementen identifizieren und geeignet behandeln lassen, entsteht *fehlertolerantes Verhalten* [Ech90] . Brutto- und Nettobetrag einer Rechnung stellen zum Beispiel eine Form von Informationsredundanz dar. Fehler in diesen Werten lassen sich allerdings erst durch die Probe gemäß ihrer Spezifikation erkennen. In diesem

Der Redundanzbereich – Lokalisierung kritischer Fehlerstellen

Der *Redundanzbereich* beschreibt den logischen Wirkungsbereich der Fehlertoleranzmaßnahmen und damit die Teile des Systems, in denen Redundanz, gleich welcher Art, koordiniert eingesetzt wird. Kritische Fehlerstellen zeigen sich – ausgehend von der Effektivität der eingesetzten Fehlertoleranzmaßnahmen – nur außerhalb der Redundanzbereiche beziehungsweise an den Übergängen zwischen diesen.

Fall erfolgt dies durch Entfernen der Mehrwertsteuer vom Bruttobetrag und dem anschließenden Vergleich mit dem Nettobetrag. Eine mögliche Reaktion im Fehlerfall kann schließlich die erneute Berechnung der Beträge aus den Einzelposten der Rechnung darstellen. Die Umsetzung von Fehlertoleranz erfordert daher, über die Redundanz hinaus, geeignete Maßnahmen zur Fehlerdiagnose und Fehlerbehandlung.

Fehlerdiagnose

Die *Fehlerdiagnose* (engl. *fault diagnosis*) umfasst zunächst die Bereitstellung geeigneter Testverfahren für die *Erkennung* von Fehlern in den redundanten Strukturelementen – im Folgenden allgemein als *Testobjekte* (engl. *test objects*) bezeichnet. Darauf aufbauend erfolgt anschließend die *Lokalisierung* des zugehörigen Fehlerbereichs. Die Bewertung der Testobjekte lässt sich dabei mithilfe der folgenden zwei Testverfahren durchführen [Ech90]:

Absoluttests (engl. *acceptance tests*) überprüfen das Testobjekt gemäß eines Soll-Ist-Vergleichs mit vorab definierten Konsistenzbedingungen. Werden diese erfüllt, ist das Testobjekt fehlerfrei, andernfalls ist es fehlerhaft. Absoluttests benötigen daher lediglich den Istzustand des Testobjekts und kommen ohne weitere Vergleichsobjekte aus, entsprechend decken sich Testobjekt und Fehlerbereich. Die erzielbare *Fehlererfassung* (engl. *fault coverage*) ergibt sich aus den Konsistenzbedingungen. Sind diese bezüglich des Sollzustands notwendig und hinreichend, ist eine zuverlässige Aufdeckung der Fehler gegeben. Die für die Codierung eingesetzten Codierungsvorschriften (siehe Kapitel 2.2.2) sind ein Beispiel für Konsistenzbedingungen, welche die Spezifikation exakt abbilden und eine vollständige Fehlererfassung ermöglichen. Im Allgemeinen steigt der Aufwand für die Formulierung vollständiger Bedingungen mit der Komplexität des Testobjekts. Dies kann im Extremfall zu einer Reimplementierung der Anwendungslogik und somit zu einer Nachberechnung des Sollzustands führen. Ist eine vollständige Abbildung der Spezifikation in Konsistenzbedingungen nicht möglich, ist lediglich eine eingeschränkte Fehlererfassung erreichbar. Zum Beispiel stehen für die Überwachung eines Sensors üblicherweise nur notwendige Bedingungen in Form von Gültigkeitsbereichen zur Verfügung. Ein fehlerhafter, jedoch gültiger Sensorwert, lässt sich somit nicht aufdecken.

Relativtests (engl. *comparison tests*) prüfen im Sinne eines Ist-Ist-Vergleichs den Zustand mehrerer Testobjekte auf Übereinstimmung. Die Zustände werden durch den Vergleich in Äquivalenzklassen eingeteilt. Die größte Klasse umfasst hierbei die als fehlerfrei angenommenen Objekte. Die anderen Klassen gelten entsprechend als fehlerhaft und bilden damit auch den Fehlerbereich. Relativtests erfassen beliebige Fehler und bieten eine vollständige Fehlererfassung, falls die Fehlerzahl die Bildung von Mehrheitsverhältnissen erlaubt. Ein bekanntes Beispiel für einen Relativtest ist der Mehrheitsentscheider bei dreifacher Redundanz. Problematisch ist der Einsatz von Relativtests bei nichtdeterministischen Prozessen, da die

Testobjekte hier auf Ähnlichkeit anstatt auf exakte Übereinstimmung geprüft werden. In diesem Fall sind wie bei einem Akzeptanztest notwendige und hinreichende Ähnlichkeitsbedingungen für die Bildung der Äquivalenzklassen erforderlich. Sind diese nicht ausreichend bestimmbar, liefern Relativtests gehäuft *Pseudofehler* (engl. *false positives*).

In der Regel lassen sich Relativtests wesentlich einfacher umsetzen, da sie nicht auf die Ableitung von Konsistenzbedingungen angewiesen sind und entsprechend konzeptbedingt eine vollständige Fehlererfassung erlauben [Ech90]. Ihre Anwendbarkeit ist jedoch aufgrund der notwendigen Vergleichszustände auf strukturelle Redundanz beschränkt. Im Gegensatz dazu können Akzeptanztests grundsätzlich für jede Art von Redundanz eingesetzt werden, sofern sich entsprechende Konsistenzbedingungen finden lassen [Ech90].

Fehlerbehandlung

Die Fehlererkennung identifiziert Fehler in den redundanten Strukturelementen, sorgt jedoch nicht für eine Unterbrechung der Fehlerausbreitung. Dies erfordert weitere Maßnahmen für die geeignete *Fehlerbehandlung* (engl. *fault handling*). Ein bekanntes Beispiel hierfür ist die wiederholte Übertragung von Nachrichten im Fehlerfall durch ein Kommunikationsprotokoll. Im Allgemeinen lassen sich die drei folgenden Arten der Fehlerbehandlung unterscheiden [ALRL04, Ech90]:

Rückwärtskorrektur (engl. *backward error recovery*) versetzt den Fehlerbereich beziehungsweise übergeordnete Komponenten in einen vorangegangenen, fehlerfreien Zustand zurück. Dies umfasst im weitesten Sinne jede Art von konsistentem Zustand, von dem aus eine spezifikationsgemäße Interaktion mit anderen Komponenten möglich ist – auch wenn dieser in der Vergangenheit nicht aufgetreten ist. Die Rückwärtskorrektur basiert auf *Rücksetzpunkten* (engl. *checkpoints*) und einer geeigneten Protokollierung der Änderungen zwischen diesen Punkten. Tritt ein Fehler auf, so wird der bereits veränderte Zustand der beteiligten Komponenten auf den letzten fehlerfreien Rücksetzpunkt zurückgesetzt. Bekannte Einsatzbereiche sind transaktionsbasierte Datenbanksysteme oder auch die bereits erwähnten Kommunikationsprotokolle. Die Rückwärtskorrektur erfordert kein dediziertes Anwendungswissen und kann daher unabhängig und transparent implementiert werden. Zudem ist auch keine strukturelle Redundanz in Form von zusätzlichen Replikaten erforderlich. Nachteilig sind die inhärenten Kosten des Ansatzes durch die notwendige Erstellung und Verwaltung der Rücksetzpunkte sowie der Zeitaufwand für die Wiederherstellung im Fehlerfall. Darüber hinaus sind lediglich Absoluttests anwendbar, was eine vollständige Fehlererfassung in der Regel deutlich erschwert.

Vorwärtskorrektur (engl. *forward error recovery*) überführt die fehlerhaften Komponenten direkt in einen konsistenten Zustand. Im Gegensatz zur Rückwärtskorrektur erfordert dies weder Wissen über die Vergangenheit noch eine Wiederholung des fehlerhaften Abschnitts. Die Herstellung des konsistenten Zustands und der Weiterbetrieb des Systems basiert entsprechend alleine auf anwendungsspezifischem Wissen und entsprechenden Fehlfunktionsannahmen. Die Einnahme eines sicheren Zustands und die Beendigung (engl. *fail-stop*) beziehungsweise der Neustart (engl. *restart*) der entsprechenden Funktion sind gängige Varianten der Vorwärtskorrektur. So lässt sich ein fehlerhafter Stellwert einer Regelungsanwendung unter Umständen dadurch beheben, dass dieser im aktuellen Schritt verworfen und die Berechnung im nächsten Schritt von einem Initialzustand von Neuem beginnt. Wie schon bei der Rückwärtskorrektur kommen Absoluttests für die Fehlererkennung zur Anwendung. Die Fehlerbehebung ist nicht transparent durchführbar, da sie maßgeblich von den Eigenschaften der Anwendung abhängt.

Maskierung (engl. *error masking/compensation*) nutzt strukturelle Redundanz, um den fehlerhaften Zustand und die betroffenen Komponenten zu verdecken. Die Maskierung erfolgt dabei durch einen Maskierer, eine zusätzliche Instanz, die stellvertretend ein fehlerfreies Ergebnis auswählt und zurückliefert. Eine bekannte Realisierung hierfür ist der Mehrheitsentscheider (Maskierer) in einem dreifach redundanten Aufbau. Der große Vorteil dieser Art der Fehlerbehandlung liegt in der Anwendbarkeit der Relativtests. Durch die vollständige Fehlererfassung ist grundsätzlich ein beliebiges Fehlverhalten der Replikate maskierbar. Da sowohl Maskierer als auch Relativtests anwendungsunabhängig sind, lässt sich die Fehlermaskierung vergleichsweise einfach implementieren. Sie vermeidet darüber hinaus den möglicherweise indeterministischen Wiederholungsbetrieb der Rückwärtskorrektur. Nachteilig ist der hohe Aufwand für die strukturelle Redundanz.

Im Kontext transienter Fehler sind die genannten Arten der Fehlerbehandlung im Allgemeinen ausreichend für die Implementierung von Fehlertoleranz. Auf eine Rekonfiguration beziehungsweise Fehlerausgrenzung, wie sie für die Behandlung permanenter Fehler notwendig ist, kann üblicherweise verzichtet werden, da transiente Fehler in den Strukturelementen nicht über die Fehlerbehandlung hinaus erhalten bleiben. Bezogen auf die Implementierung einer Fehlertoleranzmaßnahme stellt die Wahl des Testverfahrens und der Fehlerbehandlung eine maßgebliche Entwurfsentscheidung dar. So sind Relativtests und Fehlermaskierung konzeptbedingt nur auf strukturelle Redundanz anwendbar. Ebenso bedingen die anderen Redundanzarten den Einsatz von Absoluttests und Vorwärts- beziehungsweise Rückwärtskorrektur. Zusammen bestimmen die Redundanztechnik, Fehlerdiagnose und Fehlerbehandlung die Fehlertoleranzeigenschaften des jeweiligen Fehlertoleranzansatzes.

Fehlertoleranz – Redundanz, Diagnose, Behandlung

Fehlertolerantes Verhalten entsteht durch die geeignete Kombination aus Redundanztechniken und Maßnahmen zur *Fehlerdiagnose* und *Fehlerbehandlung*. Deren Auslegung bestimmt letztlich die Fehlertoleranzeigenschaften und den Umfang der Fehlererfassung.

2.2 Verwandte Arbeiten

Das Ziel dieser Arbeit ist die gezielte Beeinflussung der Zuverlässigkeit auf Ebene des Betriebssystems und ein im Sinne des zuvor skizzierten Fehlermodells umfassender Schutz sicherheitskritischer Anwendungen vor den Auswirkungen transienter Fehler. Dies bedingt zwingend den Aufbau eines lückenlosen Redundanzbereichs in Bezug auf die zu schützenden Elemente der Anwendung.

Erwartungsgemäß existiert durch die historische Entwicklung im Bereich fehlertoleranter Systeme sowie die möglichen Umsetzungsalternativen aus Redundanzart, Redundanzstrategie und Implementierungsebene eine Vielzahl von Ansätzen, die geeignet sind, transiente Fehler zu erkennen und gegebenenfalls auch zu behandeln. Bislang wurden jedoch keine Arbeiten veröffentlicht, die sich *dediziert* mit den Lücken beziehungsweise den kritischen Fehlerstellen in softwarebasierter Redundanz befassen. Vielmehr ist dies eine inhärente Eigenschaft jeder Redundanztechnik und ein wichtiger, wenn auch wenig expliziter, Aspekt bei der Implementierung von Fehlertoleranz im Allgemeinen. Die zu untersuchenden Fehlertoleranzeigenschaften definieren und verfeinern sich schrittweise, ausgehend von der Redundanzart über die Redundanzstrategie, bis hin zur konkreten Redundanztechnik. Entsprechend orientiert sich der verbleibende Teil dieses Kapitels an der bekannten Einteilung der Redundanzarten und beleuchtet die verwandten Arbeiten jeweils

ausgehend von den Gemeinsamkeiten und allgemeinen Fehlertoleranzeigenschaften bis zu den Spezifika der jeweiligen Implementierung.

Im Hinblick auf die Ziele dieser Arbeit sind vor allem die folgenden Aspekte für die weitere Betrachtung verwandter Arbeiten von Bedeutung:

Fehlertoleranzeigenschaften und Effektivität: Ermöglicht die dem Ansatz zugrunde liegende Redundanzstrategie grundsätzlich die effektive Fehlererkennung bezogen auf das Fehlermodell aus Berechnungs-, Operator- und Operandenfehlern? Nach welchem Verfahren erfolgt die Fehlerdiagnose? Wie lassen sich gegebenenfalls die hierfür notwendigen Konsistenzbedingungen ableiten? Ist die Fehlererfassung vollständig beziehungsweise welche Fehlerrate ergibt sich für die Klasse der unerkannten Datenfehler? Nach welchem Verfahren erfolgt gegebenenfalls die Fehlerbehandlung? Ist diese vollständig beziehungsweise wie hoch ist die Rate der erkannten nicht behebbaren Fehler?

Redundanzbereich: Welche Strukturelemente werden durch die Redundanz abgedeckt und welcher Wirkungsbereich ergibt sich daraus hinsichtlich der zu schützenden Regelungsanwendung? Besonderes Augenmerk liegt hierbei auf den räumlichen und zeitlichen Grenzen des Redundanzbereichs. Im Falle von Lücken ist zu klären, inwieweit aus der Partitionierung des Redundanzbereichs kritische Fehlerstellen entstehen.

Implementierung: Ein zentraler Aspekt ist die Umsetzbarkeit auf Ebene der Systemsoftware. Daraus ergeben sich die folgenden Fragen: Lässt sich die zugrunde liegende Redundanzstrategie ohne tiefgreifende Kenntnis der Anwendung beziehungsweise ohne deren Anpassung umsetzen? Ist darüber hinaus Anwendungswissen für die Fehlerdiagnose oder Fehlerbehandlung notwendig? Ist eine Implementierung der konkreten Redundanztechnik auf gängiger Hardware und ohne Ausnutzung spezifischer Hardwareeigenschaften möglich? Erfordert der Ansatz darüber hinaus Hardwareunterstützung für die Fehlerdiagnose oder Fehlerbehandlung?

2.2.1 Allgemeine Betrachtung und Abgrenzung

Grundsätzlich steht zum Schutz vor transienten Fehlern das gesamte in Abbildung 2.2 dargestellte Spektrum aus Redundanzart, Strategie und Implementierungsebene zur Verfügung. Allerdings führt nicht jede der möglichen Fehlertoleranzmaßnahmen zu geeigneten Fehlertoleranzeigenschaften hinsichtlich der zuvor eingeführten Kriterien und Ziele. Entsprechend lassen sich, wie in Abbildung 2.4 dargestellt, einige Kombinationsmöglichkeiten konzeptbedingt ausschließen.

Implementierungsebene

Ganz allgemein lassen sich sicherheitskritische Anwendungen unter Zuhilfenahme einer der Redundanzarten inhärent fehlertolerant auslegen. Dies gilt insbesondere auch für Regelungsanwendungen [ABK+93, BG97, BKL+10], welche häufig schon aufgrund der äußeren Störeinflüsse eine gewisse Robustheit gegenüber Fehlern aufweisen müssen. Obwohl diese Ansätze prinzipiell dazu geeignet sind die Klasse der transiente Fehler mit abzudecken, erweist sich die Implementierung von Fehlertoleranz auf der Anwendungsebene (vgl. Abbildung 2.4) als unzweckmäßig. Die hierfür notwendigen Änderungen an der Anwendungslogik beziehungsweise -algorithmik würden zu einem Verlust der angestrebten Anwendungstransparenz führen.

Ebenso liegen Redundanztechniken außerhalb der Betrachtung, die sich ausschließlich auf der untersten Ebene und mithilfe bestimmter Eigenschaften der Hardware implementieren lassen. *Hybride Fehlertoleranz* dient üblicherweise der Kostensenkung im Vergleich zu einer umfassenden

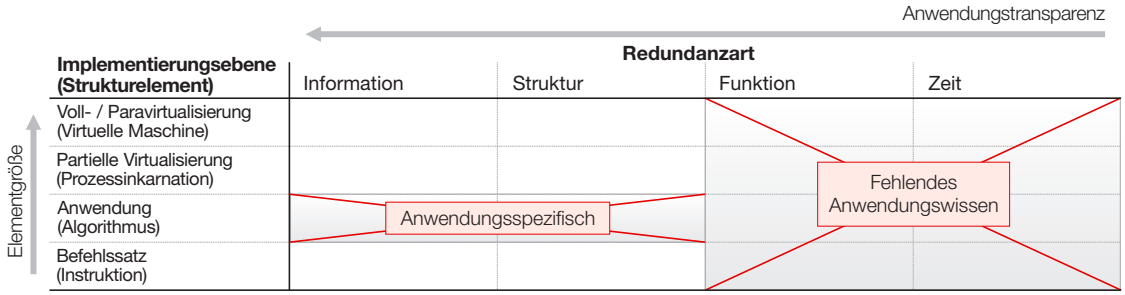


Abbildung 2.4: Für den Lösungsansatz prinzipiell unzweckmäßige Fehlertoleranzmaßnahmen.
Konzeptbedingt lassen sich bereits einige Redundanzarten und Implementierungsebenen für den verfolgten Lösungsansatz ausschließen (hervorgehobene Bereiche). Hierzu zählen im Wesentlichen all jene Fehlertoleranzmaßnahmen, die anwendungsspezifisch entworfen oder implementiert werden müssen beziehungsweise welche die Anwendung selbst betreffen und daher mit der Anforderung nach Anwendungstransparenz kollidieren.

Hardwarelösung. Bekannte Beispiele sind die IBM z900 Serie [SAC+99] oder die Flugsteuerung der Boeing 777 [Yeh96]. Entsprechend werden lediglich besonders kritische Funktionseinheiten (z. B. Speichersubsystem), Rechenkerne [SAC+99, Yeh96, HHJ90] oder auch der anfällige Hardwarekontext [GSVP03, MKR02, VPC02, RM00] durch hardwarebasierte Fehlertoleranzmaßnahmen geschützt. Diese Ansätze erleichtern und unterstützen die Einbringung zusätzlicher softwarebasierter Redundanz durch eine Abschwächung des Fehlermodells, beispielsweise durch Codierung der Datenflüsse und folglich der Vermeidung von Operatorfehlern. Hybride Ansätze nutzen diese Vereinfachungen sowohl für den Schutz von Datenflüssen [MM88, Sch11b] als auch für die Kontrollflussüberwachung [SS87, MM88, OR95, BDCDNP03, SSSF10]. Ziel dieser Arbeit ist jedoch die Berücksichtigung aller in Software sichtbaren transienten Fehler und die Abdeckung des gesamten Fehlermodells (vgl. Kapitel 2.2), weshalb keine Annahmen über die Zuverlässigkeit bestimmter Hardwarefunktionen getroffen werden können.

Funktionelle und zeitliche Redundanz

Eine auf funktioneller Redundanz beruhende Fehlererkennung erfordert Diversität in den redundanten Funktionen, wie beispielsweise die in Kapitel 2.1.2 genannte mehrfache Kontrolle eines Ventils mittels Stellungsgeber und Durchflussmengenmesser. Die entsprechende Auslegung einer sicherheitskritischen Anwendung ist daher im Allgemeinen anwendungsspezifisch und erfordert ein umfangreiches Anwendungswissen. Dies gilt insbesondere für die Fehlerdiagnose und die notwendigen Testmethoden für die Bewertung der unterschiedlichen Funktionen: Wie lange und wie viel Durchfluss ist nach dem Schließen des Ventils erlaubt? Ähnliche Probleme ergeben sich beim Einsatz zeitlicher Redundanz zur Fehlererkennung. Ob eine Anwendung in der Lage ist, Fehler durch zusätzliche Ausführungszeit zu erkennen, ist wiederum hochgradig anwendungsspezifisch.

Eine Ausnahme stellen in diesem Zusammenhang Funktionswächter für die Überwachung struktureller Elemente der Software dar, die sich unabhängig von der Anwendungssemantik umsetzen lassen [MM88]. Methoden der *statischen Analyse* [LM01] erlauben beispielsweise die Extraktion bestimmter Informationen aus dem Quell- beziehungsweise Binärprogramm. Entsprechende Fehlertoleranzansätze nutzen diese Daten [SS87, OR95, OSM02a, BDCDNP03, GRRV03, VHM03, WP05, SSSF10] für die Überwachung bestimmter Eigenschaften der Software, wie zum Beispiel Werte- und Speichergrenzen, die Gültigkeit von Referenzen oder des Kontrollflusses. Ohne Berücksichtigung der Anwendungssemantik sind die extrahierbaren Informationen jedoch beschränkt. Die entsprechenden Ansätze sind daher weder in der Lage eine vollständige Fehlererfassung noch eine lückenlose Fehlerabdeckung im Sinne des Redundanzbereichs zu erzielen [Muk08].

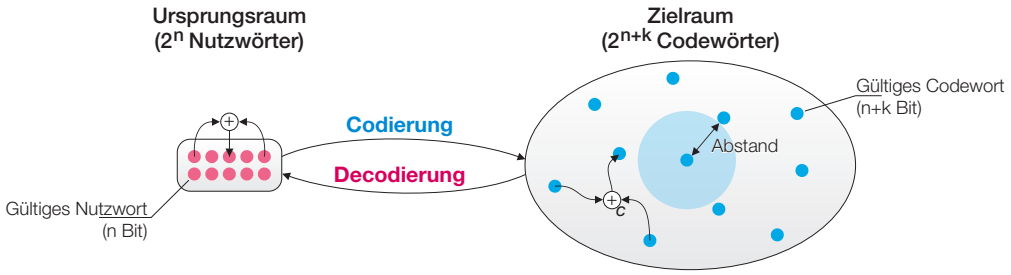


Abbildung 2.5: Grundlegendes Prinzip von Codierungsverfahren.
Die Datenwörter (n Bit) werden gemäß einer Codierungsvorschrift in Codewörter überführt. Die hierbei eingebrachte Informationsredundanz (k Bit) vergrößert dabei die Menge der möglichen Worte und ermöglicht durch den dadurch vergrößerten Abstand zwischen gültigen Codeworten die Fehlererkennung. Die inverse Decodierung der Codewörter ermöglicht die Entfernung der Redundanz und die Wiederherstellung der Ursprungsdaten. Arithmetische Codierungsverfahren erlauben zusätzlich die Durchführung arithmetischer Operationen ($\oplus_c$) direkt im Zielraum.

Zusammenfassend erweisen sich sowohl auf funktioneller, als auch auf zeitlicher Redundanz aufbauende Redundanztechniken unabhängig von ihrer Implementierungsebene als ungeeignet, da auf der Ebene der Systemsoftware kein ausreichendes Wissen über die funktionalen Zusammenhänge und das korrekte Verhalten der Anwendung zur Verfügung steht [Ech90]. Die weitere Betrachtung stützt sich daher auf Redundanztechniken, die auf struktureller Redundanz beziehungsweise auf Informationsredundanz basieren.

2.2.2 Informationsredundanz

Auch Informationsredundanz lässt sich anwendungsspezifisch einsetzen. Ein Beispiel hierfür ist die zuvor angeführte Speicherung von Brutto- und Nettowert einer Rechnung (vgl. Kapitel 2.1.2). Redundante Informationen werden in Anwendungen häufig zur Optimierung und Leistungssteigerung eingesetzt und erlauben beispielsweise in doppelt verketteten Listen ein effizientes Einfügen und Löschen von Elementen. Daneben ermöglicht ein redundanter Rückwärtszeiger jedoch auch die Erkennung von einzelnen Strukturfehlern. Entsprechend existieren verschiedene redundante Daten- und Verwaltungsstrukturen [TB82, TS86, BTM95], welche Informationsredundanz gezielt zum Zwecke der Fehlertoleranz einsetzen. Diese Ansätze sind inhärent problem- und anwendungsspezifisch und beziehen sich stets auf die zu schützende Datenstruktur und den darauf anwendbaren Operationen. Sie sind daher im Allgemeinen nicht für die Absicherung beliebiger Datenströme und Anwendungen geeignet [Ech90].

Codierung – Grundlegende Funktionsweise

Demgegenüber stellt die *Codierung* (engl. *error coding*) einen generischen Ansatz und eine allgemeine Redundanzstrategie für die Absicherung von jeglicher Art von Nutzdaten vor transienten Fehlern dar. Codierung zählt zu den gängigsten und effektivsten Fehlertoleranzverfahren und findet breite Anwendung in der Datenübertragung und -speicherung, wo die Fehlerraten der Hardware von jeher hoch sind [Ech90]. Bekannte Beispiele hierfür sind der Einsatz von ECC-Speicher in Servern oder Prüfsummen in Kommunikationsprotokollen.

Die allgemeine Funktionsweise der Codierung ist in Abbildung 2.5 dargestellt. Die *Datenwörter* (engl. *data words*) (n Bit) der Nutzdaten werden hierbei durch eine geeignete *Codierungsvorschrift* (engl. *encoding scheme*) in *Codewörter* (engl. *code words*) überführt. Im Zuge dieser Transformation werden k zusätzliche *Prüfbits* (engl. *code bits*) hinzugefügt, wodurch die Menge der möglichen

Codewörter auf 2^{n+k} ansteigt. Die Anzahl der gültigen Codewörter entspricht jedoch weiterhin der Zahl der Datenwörter (2^n), wodurch ein Abstand zwischen gültigen Codewörtern entsteht. Umgekehrt ermöglicht die inverse Decodierung die Entfernung der redundanten Information und die Wiederherstellung der ursprünglichen Datenwörter.

Codierung – Allgemeine Fehlertoleranzeigenschaften

Unabhängig von der konkreten Codierungsvorschrift basiert die Fehlererkennung auf der dünneren Besetzung des Zielraums und dem vergrößerten Abstand zwischen gültigen Codewörtern, wie in den Abbildungen 2.5 und 2.6 dargestellt. Mutationen durch transiente Fehler führen daher mit hoher Wahrscheinlichkeit zu einem ungültigen Codewort, welches entsprechend erkannt werden kann. Die Fehlerdiagnose erfolgt hierbei durch Absoluttests, welche die Konformität mit den Codierungsvorschriften prüfen. Die hierbei zur Verfügung stehenden Testbedingungen sind sowohl notwendig als auch hinreichend und erlauben daher die zuverlässige Aufdeckung von Fehlern. Solange ein transienter Fehler ein gültiges Codewort nicht wiederum in ein anderes, gültiges Codewort überführt, ist die Fehlererfassung daher effektiv und vollständig.

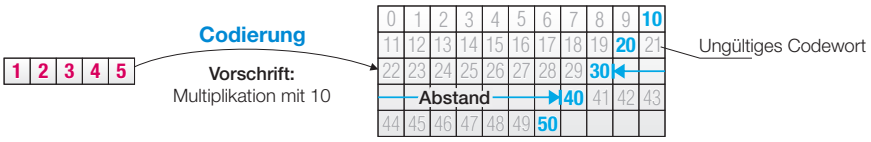


Abbildung 2.6: Beispiel einer Codierungsvorschrift.
Die Codierungsvorschrift *Multipliziere mit 10* überführt die Zahlen Eins bis Fünf in ihre codierte Form [10, 20, 30, 40, 50]. Der Abstand im Dezimalsystem ist entsprechend stets Zehn, wodurch sich der Coderaum folglich auf das Zehnfache vergrößert. Codewörter, die sich nicht ganzzahlig durch Zehn teilen lassen, werden als ungültig erkannt.

Die gewählte Redundanzstrategie, respektive Codierungsvorschrift, bestimmt über die Besetzung des Zielraums sowie den konkreten Abstand der entstehenden Codewörter untereinander. Abbildung 2.6 veranschaulicht diesen Zusammenhang am Beispiel einer einfachen Codierungsvorschrift: Der resultierende Abstand in der dezimalen Darstellung beträgt hier stets zehn. Die Art und Schwere der erkennbaren Bitfehler ergibt sich weiterhin aus der Vorschrift selbst. Für ein Paritätsbit wird beispielsweise die Anzahl der mit Eins belegten Datenbits bestimmt und die Information *gerade* oder *ungerade* in das Prüfbit codiert – der resultierende Abstand beträgt hier nur zwei Bit (gerade/ungerade + Prüfbit). Jedoch lassen sich folglich alle ungeraden Bitfehler erkennen, während eine gerade Anzahl an Fehlern unerkannt bleibt.

Eine weitere Eigenschaft der Strategie ist die Darstellung der Codewörter sowie die darauf anwendbaren *Operationen* wie Codierung, Decodierung oder arithmetische Operatoren. Mit Blick auf die Implementierung lassen sich die k Prüfbits und n Datenbits auf unterschiedlichen Wegen zu einem Codewort zusammenfügen. Es werden im Allgemeinen die in Abbildung 2.7 dargestellten Varianten unterschieden [Rao74]:

Systematisch (engl. *systematic code*)**:** Eine Codierung, deren Codewörter aus $n+k$ Bits bestehen, ist *systematisch*, wenn die Nutzinformation durch n und die Prüfinformation durch k Stellen eines Codewortes getragen werden (vgl. Abb. 2.7 oben links). Die Zuordnung der Stellen ist dabei eindeutig und unabhängig von den Eingabedaten. Diese Darstellung hat den Vorteil, dass ein direkter Zugriff auf den funktionalen Anteil weiterhin möglich und entsprechend keine explizite Decodierung erforderlich ist.

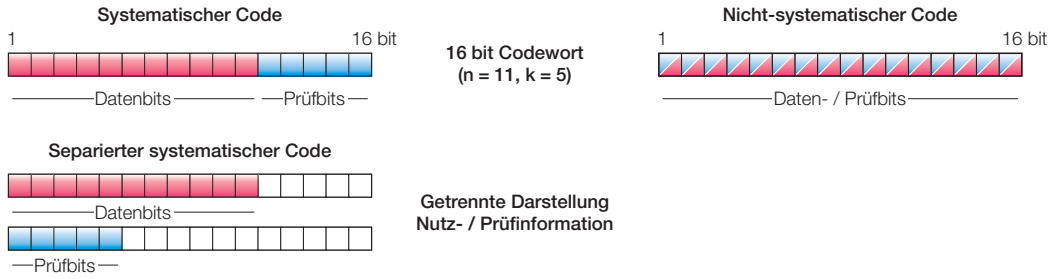


Abbildung 2.7: Entwurfsalternativen für die Darstellung von Codewörtern.
Codewörter setzen sich aus *n* Daten- und *k* Prüfbits zusammen, wobei ihre Darstellung von der verwendeten Codierungsvorschrift abhängt. Bei *nicht-systematischen* Codes existiert keine Zuordnung zu dedizierten Bitpositionen, weshalb auch kein direkter Zugriff auf die Daten möglich ist. Demgegenüber sind die Daten bei *systematischen* Codes ohne Decodierung lesbar. Ist der Code weiterhin *separiert*, erfolgt die Speicherung und Verarbeitung von Daten und Prüfinformationen getrennt.

Demgegenüber kann auf die Daten bei *nicht-systematischer* Codierung (vgl. Abb. 2.7 oben rechts) nicht ohne vorherige Decodierung zugegriffen werden. Eine dedizierte Zuordnung der $n+k$ Stellen zu der Nutz- beziehungsweise Prüfinformation ist nicht möglich.

Separiert (engl. *separate code*)**:** Bei *separierter* Codierung (vgl. Abb. 2.7 unten links) ist das Codewort ein 2-Tupel, bestehend aus den funktionalen Daten (n Bit) und der Prüfinformation (k Bit). Arithmetische Operationen auf den Codewörtern erfordern in diesem Fall zwei separate Berechnungsschritte für die beiden Teile des Tupels. Entsprechend ist eine separierte Codierung stets auch systematisch.

Auf der anderen Seite kann die Berechnung bei *nicht-separierter* Codierung in einer kombinierten Operation und in einem Schritt erfolgen. Eine getrennte Verarbeitung der Nutz- und Prüfinformationen ist nicht notwendig. Die Codewörter können hier systematisch beziehungsweise nicht-systematisch vorliegen.

Die Anordnung der Prüf- und Nutzinformation ist eine Eigenschaft der jeweiligen Redundanzstrategie. Ein Paritätsbit lässt sich beispielsweise nur systematisch ablegen, da seine Position im Codewort für die Decodierung vorab bekannt sein muss. Grundsätzlich hat die Darstellung keinen direkten Einfluss auf die Effektivität der Codierung, sie ist jedoch mittelbar für die Bestimmung des Redundanzbereichs relevant. So ist bei separierten Codes zu bedenken, dass Nutz- und Prüfinformation an getrennten Stellen gespeichert sein können und deren Verarbeitung darüber hinaus einen potenziell größeren zeitlichen Versatz aufweist. Allgemein ist in diesem Zusammenhang die systematische, nicht-separierte Codierung attraktiv, da deren Operationen jeweils beide Teile des Codewortes umfassen und darüber hinaus die Decodierung entfallen kann.

Codierung

Die Fehlererkennung beruht bei der *Codierung* von Daten auf dem Abstand zwischen gültigen Codewörtern. Verschiedene Strategien unterscheiden sich im Wesentlichen hinsichtlich der *Codierungsvorschrift*. Diese beschreibt die Überführung von Datenwörter in Codewörter eindeutig und erlaubt daher eine vollständige Fehlererfassung durch Absoluttests.

Datencodierung

Bei den bisher als Beispiel genannten Paritätsbits [Ham50] beziehungsweise der zyklischen Redundanzprüfung [Sha49] handelt es sich um Verfahren der reinen *Datencodierung*. Sie sind effektiv hinsichtlich der erzielbaren Fehlertoleranz gegenüber transienten Fehlern, jedoch konzeptionell auf Datenflüsse limitiert. Die zur Verfügung stehenden Operatoren beschränken sich auf die Codierung beziehungsweise Decodierung der Daten und erlauben nicht die Durchführung von Berechnungen auf Codewörtern. Folglich müssen Codewörter decodiert und die Nutzdaten entnommen werden, bevor diese einer weiteren Verarbeitung zugeführt werden können. Die Datencodierung ist daher nicht in der Lage das Fehlermodell aus Kapitel 2.1.1 abzudecken und beispielsweise Operator- oder Berechnungsfehler effektiv zu erkennen. Zudem beschränkt sich der Redundanzbereich auf die Verweildauer der Daten im Speicher bis zu ihrer Verwendung.

Arithmetische Codierung

Im Gegensatz dazu erlaubt die *Arithmetische Codierung*[1] (engl. *arithmetic coding*) die Durchführung von Berechnung auf Codewörtern ohne deren vorherige Decodierung beziehungsweise den Verlust der Schutzeigenschaften [PW72, Rao74]. Sie unterscheidet sich von der reinen Datencodierung durch die Bereitstellung zusätzlicher *arithmetischer Operatoren* ($\bigcirc_c$), welche direkt auf den Codewörtern des Zielraums arbeiten. Abbildung 2.5 zeigt dies exemplarisch anhand einer codierten Addition ($+_c$) zweier gültiger Codewörter und der Rückgabe eines wiederum gültigen Codewortes als Ergebnis. Eine fehlerhafte Ausführung dieser codierten arithmetischen Operationen führt, analog der Mutation eines Codewortes, mit hoher Wahrscheinlichkeit zu einem ungültigen Codewort als Ergebnis. Arithmetische Codierung ist daher grundsätzlich geeignet, transiente Fehler in der eigentlichen *Programmausführung* zu erkennen und damit potenziell das gesamte Fehlermodell abzudecken.

Aufgrund ihrer Eigenschaften sind arithmetische Codierungsverfahren von hoher Relevanz für die in dieser Arbeit angestrebten Ziele. Wie schon bei der Datencodierung existieren für die Umsetzung von arithmetischer Codierung verschiedene Strategien und Codierungsvorschriften. Diese unterscheiden sich vornehmlich in den unterstützten arithmetischen Operatoren, dem erzielbaren Redundanzbereich sowie der tatsächlichen Abdeckung des Fehlermodells. Tabelle 2.1 gibt einen Überblick über relevante und in Software umsetzbare Varianten sowie deren Eigenschaften.

Generell gilt für arithmetische Codierung, dass sie auf ganzzahlige Arithmetik beschränkt ist. Es lassen sich entsprechend auch nur ganzzahlige Werte codieren, wobei sich andere Darstellungen zumeist in Software nachbilden lassen [Sch11b]. Für die Realisierung einer effektiven Fehlertoleranz ist zudem die Codierung der *gesamten* zu schützenden Anwendung erforderlich. In der Folge müssen entsprechend *alle* sicherheitskritischen Berechnungen lückenlos durch codierte Operatoren darstellbar sein. Für alle Varianten gelten die folgenden gemeinsamen Eigenschaften beziehungsweise Bezeichner:

- v bezeichnet den funktionalen, *uncodierten Wert* (Datenwort) beziehungsweise die *Variable* mit dem entsprechenden Bezeichner.
- A repräsentiert die für die Codierung verwendete Konstante, den *Codierungsschlüssel*.
- v_c bezeichnet die codierte Repräsentation (Codewort) der Variablen v, im Folgenden auch *codierter Wert* genannt.
- $\bigcirc_c$ stellt die codierte Variante einer arithmetischen Operation ($+$, $-$, $\cdot$, $/$) dar. Diese erlaubt die Durchführung der Berechnung auf codierten Werten, zum Beispiel die Addition $z_c = x_c +_c y_c$.

[1] In der Literatur selten auch *Homomorphe Codierung* (engl. *homomorphic encoding*) genannt.

	Fehlererkennung					Unterstützte Operatoren (codiert)							Redundanzbereich	
			Operandenfehler											
Code	Berechnungsfehler	Operatorfehler	Datenfehler	Adressfehler	Verlorener Zugriff	Addition	Subtraktion	Multiplikation	Division	Bitweise Operatoren	Logische Operatoren	Vergleichsoperatoren	Datenfluss	Kontrollfluss
Residue	●	◐	●	◐	○	●	●	●	○	○	●	○	◐	○
AN	●	○	●	○	○	●	●	●	●	○	●	○	◐	○
ANB	●	●	●	●	○	●	●	●	○	○	●	●	◐	●
ANBD	●	●	●	●	●	●	●	●	○	○	●	●	●	●

● Möglich ◐ Unvollständige Fehlererkennung ○ Nicht möglich

Tabelle 2.1: Übersicht über die gängigsten Varianten der arithmetischen Codierung.
Die verschiedenen Strategien *arithmetischer Codierung* unterscheiden sich im Wesentlichen in der möglichen Fehlererkennung, den unterstützten Operatoren sowie dem entstehenden Redundanzbereich bezogen auf Daten- und Kontrollfluss.

Residue-Code

Residue-Codes (dt. *Restbetragcode*) stellen eine sehr grundlegende Form der arithmetischen Codierung dar und sind eine der wenigen Varianten im praktischen Einsatz, welche Codewörter systematisch und separiert darstellen [AGM+71]. Das Codewort v_c wird entsprechend durch ein Tupel aus dem Wert v und dem Restbetrag R repräsentiert. Die Codierung ist dabei definiert als:

$$v_c = (v, v \bmod A) = (v, R) \quad \text{mit} \quad A > 1 \tag{2.1}$$

Die zur Verfügung stehenden Operatoren sind in Tabelle 2.1 aufgezeigt. Von entscheidendem Nachteil ist hier die fehlende Unterstützung für Vergleichsoperatoren: Ein Vergleich zweier Codeworttupel erfordert stets auch den Vergleich der ungeschützten funktionalen Werte v und ist entsprechend anfällig für transiente Fehler. Folglich ist eine Codierung von Kontrollflussanweisungen im Allgemeinen ausgeschlossen. Residue-Codes sind zwar in der Lage Daten- und Berechnungsfehler vollständig zu erkennen, alle weiteren Fehlerklassen werden jedoch gar nicht oder nur teilweise abgedeckt.

Insgesamt haben Residue-Codes den Vorteil der systematischen Darstellung der Codewörter. Jedoch sind die lückenhafte Fehlererkennung und der beschränkte Redundanzbereich von großem Nachteil für die erzielbaren Fehlertoleranzeigenschaften. Bekannte Implementierungen [AGM+71, NN99, SKK+08] beschränken sich auf Hardwarelösungen, wobei eine softwarebasierte Realisierung grundsätzlich möglich ist [Sch11b].

AN-Code

Die Codierung mittels *AN-Codes* ist ebenfalls vergleichsweise simpel und beruht auf der namensgebenden Multiplikation von Codierungsschlüssel A mit einer Zahl N (engl. *number*), in diesem Fall der Variablen v. Gültige Codewörter sind entsprechend stets Vielfache von A und definiert durch:

$$v_c = A \cdot v \quad \text{mit} \quad A > 1 \tag{2.2}$$

Im Gegensatz zu den Residue-Codes sind AN-Codes nicht-systematisch [AGM+71]. Sie unterstützen alle relevanten arithmetischen Operatoren inklusive der Division. Ebenso ist ein direkter Vergleich zweier codierter Werte grundsätzlich möglich, allerdings ist das Ergebnis ($>$, $<$, $=$) weiterhin uncodiert und damit wiederum anfällig gegenüber transienten Fehlern. Entsprechend sind auch AN-Codes nicht in der Lage codierte Vergleichsoperatoren im eigentlichen Sinne bereitzustellen und damit auch die Codierung von Kontrollflussentscheidungen zu ermöglichen. Die Abdeckung des Fehlermodells beschränkt sich auf Berechnungs- und Datenfehler. Fehlerhafte oder verlorene Zugriffe sind ebenso wenig erkennbar wie vertauschte Anweisungen (vgl. Tabelle 2.1).

Es existieren verschiedene auf AN-Codes basierende Ansätze zum softwarebasierten Schutz vor transienten Fehlern. EC-AN [FSS09] ist ein auf der *Low-Level Virtual Machine* (LLVM) [LA04] aufbauendes Werkzeug zur Transformation ganzer Programme. Bei ED4I [OMM02] erfolgt die Codierung ebenfalls im Zuge einer werkzeuggestützten Transformation der sicherheitskritischen Anwendung. Hierbei werden sowohl Daten als auch Instruktionen dupliziert und mit einem festen A codiert. Die Berechnung ist entsprechend korrekt, falls sich die Ergebnisse der beiden Verarbeitungsstränge um Vielfache von A unterscheiden. Bei ED4I handelt es sich um einen gemischten Redundanzansatz, der in Teilen auch auf struktureller Redundanz beruht. Insgesamt weist ED4I jedoch eine unvollständige Fehlererfassung auf, da beispielsweise Bitfehler in ungenutzten höherwertigen Bits nicht zuverlässig erkannt werden können. In ähnlicher Weise basiert TRUMP [CRA06] grundsätzlich auf der dreifachen Ausführung der Anwendung (vgl. Kapitel 2.2.3). Der Ansatz ersetzt jedoch ein Replikat durch die Einbringung von AN-Codes in die replizierte Ausführung. Die Codierung des Replikats erfolgt hierbei jedoch nur partiell und beschränkt sich beispielsweise auf Registerinhalte. Entsprechend zeigt sich die so geschützte Anwendung weiterhin anfällig gegenüber transienten Fehlern.

ANB-Code

Zusammenfassend erlauben Residue- wie auch AN-Codes die Absicherung einzelner Rechenschritte und klären die Frage, ob *richtig gerechnet* wurde. Darüber hinaus weisen sie jedoch Lücken hinsichtlich der Identität beziehungsweise der Reihenfolge der Daten und Anweisungen auf. Zur Lösung dieses Problems nutzt Forin [For89] statische Signaturen (B). Die so erweiterten *ANB-Codes* fügen der Codierung eine für jede Variable v eindeutige Signatur B_v hinzu:

$$v_c = A \cdot v + B_v \quad \text{mit} \quad 0 < B_v < A \tag{2.3}$$

Durch die Einführung der Signaturen sind ANB-Codes in der Lage die Identität der Daten festzustellen. Über das Berechnungsergebnis lässt sich mittelbar auch die Identität des arithmetischen Operators überprüfen. So weist eine Multiplikation eine andere Ergebnissignatur auf als eine Addition. Somit lassen sich auch vertauschte Daten und Operationen erkennen und die Frage klären, ob *das Richtige gerechnet* wurde.

ANB-Codes büßen die Unterstützung der Division aufgrund der komplexen Signaturberechnung gegenüber den AN-Codes ein – im Bedarfsfall lässt sich diese jedoch aufwendig in Software nachbilden [Sch11b]. Im Gegenzug erlauben die Signaturen die Bereitstellung codierter Vergleichsoperatoren und entsprechend die Erweiterung des Redundanzbereichs auf den Kontrollfluss. Die einzige Einschränkung in diesem Zusammenhang stellen verlorene Variablenzugriffe beziehungsweise veraltete Daten dar, die sich nach wie vor nicht erkennen lassen und den Redundanzbereich für Datenflüsse weiterhin beschränken.

ANBD-Code

Um die verbliebene Lücke durch verlorene Zugriffe zu schließen und damit eine vollständige Fehlererfassung im Sinne des Fehlermodells aus Kapitel 2.1.1 zu erreichen, führt Forin [For89] zusätzlich Zeitstempel (D) zur Aufdeckung veralteter Daten ein. Der resultierende *ANBD-Code* stellt damit die bislang vollständigste Redundanzstrategie zur arithmetischen Codierung dar:

$$v_c = A \cdot v + B_v + D \quad \text{mit} \quad 0 < B_v + D < A \tag{2.4}$$

ANBD-Codes wurden erstmals für die Implementierung von Fehlertoleranz in Schienenverkehrssystemen [HG93] genutzt. Das von Forin vorgestellte Konzept des *Vital Coded Processor* (VCP) [For89] stellt hierbei eine hybride Lösung dar. Die Anwendungssoftware wird zunächst auf programmiersprachlicher Ebene in eine codierte Variante überführt. Ein spezifischer Hardwareentwurf ermöglicht im Anschluss die direkte Verarbeitung und die Kommunikation von codierten Datenströmen und damit einen potenziell vollständigen Redundanzbereich über das gesamte Rechensystem. Leider existieren keine Daten zu der hierdurch erzielbaren Effektivität.

Software Encoded Processing (SEP) [WF07] basiert auf der Interpretation und Transformation der sicherheitskritischen Anwendungen zur Laufzeit, wobei der Interpreter selbst wiederum durch eine ANBD Codierung geschützt ist. Der Ansatz weist aufgrund der Dynamik sowie der komplexen Implementierung einen eingeschränkten Redundanzbereich auf und erfordert konzeptbedingt eine vertrauenswürdige *Prüfinstanz* (engl. *checker*) für die Decodierung. Darüber hinaus vervielfachen sich die Laufzeitkosten durch Interpretation und Codierung um einen Faktor von bis zu 10^5.

Compiler Encoded Processing (CEP) [SSSF10, Sch11b] stellt eine auf ANBD-Codes basierende Weiterentwicklung von EC-AN dar. Wie dieser basiert CEP auf der LLVM und überführt C-Quellcode direkt in eine codierte Binärdatei. Im Gegensatz zu SEP verbessert sich der Redundanzbereich zwar durch die statische Codierung zur Übersetzungszeit, jedoch ist weiterhin eine vertrauenswürdige Prüfinstanz notwendig. Messungen konnten entsprechend ein gewisses Maß an SDCs nachweisen. Auch hier scheint, neben der Prüfinstanz, die Komplexität des Ansatzes eine weitere Quelle für Lücken im Redundanzbereich darzustellen [Sch11b]. Die induzierten Laufzeitkosten sind im Vergleich mit SEP weit weniger dramatisch und von bis zu einem Faktor 10^3.

> **Arithmetische Codierung – ANBD-Codes**
>
> Arithmetische Codierung erweitert die Fehlererfassung von der reinen Datenintegrität hin zu den eigentlichen Berechnungen. ANBD-Codes stellen in diesem Zusammenhang die vollständigste Ausprägung arithmetischer Codierung dar und erlauben konzeptionell die Erkennung aller Operator-, Operanden- und Berechnungsfehler. Der Aufbau eines geschlossenen Redundanzbereichs erfordert allerdings die lückenlose Codierung *aller* Berechnungen und Daten.

Codierung – Fehlertoleranz

Insgesamt stellt Codierung in erster Linie einen Mechanismus zur Fehlererkennung dar. Die Fehlerdiagnose erfolgt letztlich durch einen Absoluttest am Ende der codierten Berechnung beziehungsweise an bestimmten Prüfpunkten (SEP, CEP). Insbesondere bei der nicht-systematischen Darstellung ist hierfür eine Decodierung des zu prüfenden Codewortes notwendig. Die Fehlerlokalisierung bezieht sich jeweils auf den Berechnungsabschnitt zur letzten Prüfung. Die weitergehende Fehlerbehandlung erfordert indes zusätzliche Maßnahmen. Grundsätzlich ist eine Vorwärtskorrektur mittels fehlerkorrigierender AN-Codes [Man67, Rao70, Avi71, MGO72] denkbar, deren

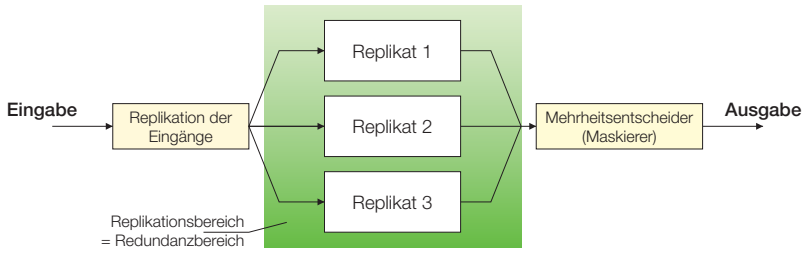


Abbildung 2.8: Strukturelle Redundanz in Form einer replizierten, dreifach redundanten Ausführung (TMR).
Beispiel einer redundanten Ausführung mit drei Replikaten. Die Eingabe wird zunächst geeignet repliziert und aufgeteilt. Nach der Ausführung der Replikate werden die Ergebnisse an den Mehrheitsentscheider übergeben. Im Falle einer qualifizierenden Mehrheit, in diesem Fall 2-von-3, wird diese als Ausgabe weitergegeben. Andernfalls signalisiert der Mehrheitsentscheider einen Fehler. Der Redundanzbereich ist im Falle der Replikation identisch mit dem Replikationsbereich nach [RM00].

Korrektureigenschaften entsprechen jedoch denen der normalen Datencodierung. Entsprechend lassen sich konzeptbedingt lediglich korrumpierte Daten und damit die Klasse der Operatorfehler behandeln. Eine rein softwarebasierte Realisierung erscheint nach Untersuchungen von Schiffel [Sch11b] zudem ausgeschlossen. Daher bleibt letztlich nur die Rückwärtskorrektur und erneute Durchführung von fehlgeschlagenen Berechnungen.

2.2.3 Strukturelle Redundanz und Replikation

Wie bereits skizziert, führt der koordinierte Einsatz struktureller Redundanz zu *Replikation*. Diese Redundanzstrategie zählt wie die Codierung zu den weitverbreiteten und gängigen Fehlertoleranzmaßnahmen – daher wird Redundanz häufig mit struktureller Redundanz gleichgesetzt [Ech90].

Grundlegende Funktionsweise

Die grundlegende Funktionsweise der Replikation ist am Beispiel der dreifachen Redundanz (TMR) in Abbildung 2.8 dargestellt. Der Aufbau eines replizierten Systems erfolgt dabei, unabhängig von der Implementierungsebene, stets nach dem gleichen Muster: Die Strukturelemente (Prozessinkarnationen, Algorithmen, Instruktionen) werden vervielfältigt und ergeben die zugehörigen Replikate. Diese bilden gemeinsam den Redundanzbereich, innerhalb dessen ein Fehlverhalten einzelner Replikate erkannt und maskiert werden kann. Die notwendigen Eingaben werden dabei vor der Ausführung der Replikate vervielfältigt und auf diese verteilt. Ebenso erfolgt im Anschluss ein Vergleich der Replikatausgänge mithilfe eines *Mehrheitsentscheiders*. Die Ausgabe ist schließlich, im Falle eines Quorums, das Ergebnis der qualifizierenden Mehrheit.

Da voneinander abweichende Ergebnisse durch den Relativtest konzeptbedingt als Fehlverhalten eingestuft werden, ist im Allgemeinen ein deterministisches Verhalten der einzelnen Replikate erforderlich [KDK+89, Ech90]. Dieser *Replikdeterminismus* (engl. *replica determinism*) [Pol94] ist im Falle der hardwarebasierten synchronen Replikation (engl. *lockstepping*) durch die Ausführung der Replikate im Gleichtakt gegeben. Bei der in Software realisierbaren asynchronen Replikation (engl. *redundant multithreading*) können die Eingaben durch den möglichen zeitlichen Versatz voneinander abweichen. Liest die Anwendung beispielsweise den Wert eines Zeitgebers, so wird die replizierte Ausführung mit hoher Wahrscheinlichkeit zu unterschiedlichen Zeitwerten und damit potenziell zu korrekten jedoch abweichenden Ergebnissen führen. Um dies zu vermeiden, ist daher eine *geeignete* und *konsistente* Replikation der Eingänge zwingend erforderlich [Kop97].

Im Kontext softwarebasierter Fehlertoleranz wird die mehrfache Ausführung von Instruktionen oder Instruktionsfolgen teilweise auch der zeitlichen Redundanz zugerechnet und als *Temporal*

oder *Time Redundancy* bezeichnet [GRRV06a]. Dieser Betrachtung folgend werden nicht die Strukturelemente, sondern die Ausführungszeit repliziert und die Fehlererkennung entsprechend der temporalen Diversität zugerechnet. Im Rahmen dieser Arbeit wird die replizierte Ausführung unabhängig von der konkreten Implementierung als strukturelle Redundanz angesehen.

Allgemeine Fehlertoleranzeigenschaften

Die Fehlererkennung beruht bei der Replikation, im Gegensatz zur Codierung, alleine auf dem Vergleich der Ausgaben der redundanten Ausführung. Entscheidend ist das Fehlverhalten an den Außengrenzen der Replikate, welches sich im Zuge eines Relativtests aufdecken lässt. Der entstehende Redundanzbereich entspricht folglich dem Replikationsbereich (vgl. Abbildung 2.8) und ergibt sich aus den vervielfältigten Strukturelementen. Die erzielbare Fehlererfassung im Rahmen der Replikation ist vollständig im Sinne des Fehlermodells, *sofern* eine Fehlerausbreitung zwischen Replikaten ausgeschlossen werden kann. In diesem Fall lassen sich sowohl Kontroll- als auch Datenflussfehler aufdecken.

Über die erzielbaren Fehlertoleranzeigenschaften der Replikation entscheidet alleine das Maß an Redundanz respektive die Anzahl der Replikate – im Gegensatz zu den komplexen Zusammenhängen bei den Codierungsvorschriften. Ohne weitere Annahmen an das Fehlverhalten (siehe Kapitel 3.2.1), lassen sich durch einen Vergleich der Ergebnisse von zwei Replikaten (DMR) lediglich Einzelfehler erkennen. Ein drittes Replikat und dreifache Redundanz (TMR) ermöglicht darüber hinaus sowohl Fehlerdiagnose als auch die Fehlerbehandlung durch Maskierung des fehlerhaften Replikats. Im Allgemeinen lassen sich f Fehler mittels $2 \cdot f + 1$ Replikaten maskieren.

> **Replikation**
>
> *Replikation* basiert auf der redundanten Ausführung von Strukturelementen. Abhängig von der Anzahl der Replikate ermöglicht dies die Fehlererkennung oder sogar die Fehlerbehandlung durch Maskierung. Die Fehlererfassung durch den Relativtest der Ergebnisse ist dabei vollständig, sofern die Fehlerausbreitung zwischen den Replikaten ausgeschlossen ist.

Replikation durch Virtualisierung der Maschine

Zur Erzielung eines möglichst vollständigen Redundanzbereichs ist ein naheliegender Ansatz die ebenso vollständige Replikation des gesamten Softwaresystems bestehend aus Betriebssystem und Anwendungen. Entsprechende Techniken der Voll- beziehungsweise Paravirtualisierung nutzen *virtuelle Maschinen* (VM) (engl. *virtual machine*) zur Darstellung der Replikate, weshalb diese Ansätze häufig unter dem Begriff *Redundant Virtual Machines* (RVM) geführt werden [Muk08]. Das grundsätzliche Vorgehen entspricht der asynchronen Hardwarereplikation und umfasst üblicherweise mehrere Rechensysteme, wobei die Aufteilung der Eingangssignale sowie der Vergleich der Ausgangssignale durch den *virtuellen Maschinen Monitor (VMM)* (engl. *virtual machine monitor*) [RG05] anstelle der üblichen Spezialzweckhardware erfolgt [BS96, TLM+09, RHKSP06]. Die notwendige Synchronisierung der Replikate wird durch eine Aufteilung der Ausführung in *Epochen* (engl. *epoch*) erreicht. Signalreplikation und Vergleich finden jeweils an den Bereichsgrenzen der Epochen statt. Ansätze wie ExtraVirt [LRC05] oder das kommerzielle EverRun [TLM+09] erfordern keine redundanten Rechensysteme, sondern nutzen einzelne Kerne in Mehrkernsystemen für die Ausführung der Replikate. TFT [Bre98] stellt weiterhin eine auf Paravirtualisierung aufbauende generische Replikationsschicht dar, welche sich an beliebige VMMs beziehungsweise Betriebssysteme anpassen lässt.

RVMs haben den Vorteil des größtmöglichen Redundanzbereichs, welcher das gesamte Softwaresystem umfasst. Darüber hinaus kann die Replikation transparent für Anwendungen und Betriebssystem durchgeführt werden. Allerdings stellen die gängigen Verfahren eine hybride Lösung dar und erfordern weiterhin ein gewisses Maß an Hardwareredundanz. Zudem bietet die Replikation auf Systemebene praktisch keinerlei Selektivität und induziert durch das einheitliche Schutzniveau unnötig hohe Kosten. RVMs erweisen sich daher für die Umsetzung der angestrebten Ziele als wenig geeignet.

Aufgrund ihres Einsatzgebietes sind RVMs von Ansätzen der *byzantinischen Fehlertoleranz* (engl. *byzantine fault tolerance*) zu unterscheiden. Eine Abgrenzung zu dieser Klasse von Systemen liefert die Fehlerhypothese in Kapitel 3.2.1.

Replikation auf der Betriebssystemebene (Partielle Virtualisierung)

Eine weit flexiblere Dimensionierung des Redundanzbereichs erlaubt eine auf der *partielle Virtualisierung* (engl. *partial virtualisation*) [CMDD62] der Hardware durch das Betriebssystem aufbauende Replikation. Die hierdurch ermöglichte räumliche wie zeitliche Isolation von *Prozessen* (engl. *process*) erlaubt die Kapselung der einzelnen Replikate durch die Bereitstellung von getrennten Ausführungskontexten und Adressräumen. Die Abbildung der Anwendung auf entsprechende *Prozessinkarnationen* stellt einen inhärenten Schritt innerhalb des regulären Systementwurfs dar. In der Folge lassen sich die zu replizierenden Teile üblicherweise einfach identifizieren [SMR+07]. Die geeignete und konsistente Replikation der Daten und Signale gewinnt gegenüber dem pauschalen Ansatz der RVM an Bedeutung. Dies betrifft beispielsweise die Nutzung von Systemdiensten oder Hardwareperipherie. Die Synchronisation erfolgt wiederum an definierten Grenzen, üblicherweise zu Beginn und Ende der Ausführung. Ein häufig genutztes Konzept für die Realisierung der Replikation auf Betriebssystemebene ist der Einsatz einer Replikationsschicht [Bre98, AVFK02, SMR+07, SBM+09, DH12]. Alternativ lassen sich die notwendigen Funktionen beispielsweise auch durch Techniken der *aspektorientierten Programmierung (AOP)* [EFB01] einbringen [ASB+08]. Um den Mehraufwand für die Prozessverwaltung durch das Betriebssystem einzusparen, existieren weiterhin verschiedene Ansätze [RM00, WKWY07], welche die spezifischen Eigenschaften von SMT-fähigen Prozessoren (engl. *simultaneous multithreading, SMT*) zur Fehlerisolation ausnutzen, um die Replikation mittels Fäden (engl. *thread*) zu realisieren. Diese Ansätze sind somit auf bestimmte Hardwarefunktionen angewiesen und zielen im Wesentlichen auf eine Optimierung der Leistungsfähigkeit ab.

Aufgrund der direkten Korrelation zwischen sicherheitskritischer Anwendung und replizierbaren Prozessinkarnationen weist die Replikation auf Ebene des Betriebssystems sowohl ein hohes Maß an Selektivität als auch eine passgenaue Abstimmung des Redundanzbereichs auf. Darüber hinaus lassen sich die Replikate in den regulären Entwurfsprozess des Systems integrieren und beispielsweise in der Ablaufplanung berücksichtigen. Demgegenüber konnten Messungen [AVFK02] jedoch auch Lücken in dem entstehenden Redundanzbereich nachweisen, die sich vorwiegend aus der Signalreplikation und den Mehrheitsentscheidern ergeben.

Replikation auf der Befehlssatzebene

Transiente Fehler zeigen sich aus Sicht der Software initial in den Instruktionen der Befehlssatzebene. Eine Replikation dieser Strukturelemente stellt somit den kleinsten gemeinsamen Nenner und die erste mögliche Verteidigungslinie dar. Der Redundanzbereich ergibt sich in diesem Fall aus den replizierten Instruktionen beziehungsweise *Verarbeitungsströmen* (engl. *processing stream*). Die Replikation kann dabei prinzipiell auf beliebige Teile der Software angewendet werden und einzelne Berechnungen, Funktionen, Anwendungen oder auch das gesamte System umfassen.

Original Code

```
// Vordefiniert:
// r2 (Wert),r4 (Adresse)
ld  r3 = [r4]
add r1 = r2,r3
st  [r1] = r2

/* Adressberechung:
   r2 wird an die Stelle
   r2+r3 geschrieben.  */
```

EDDI

```
// r2,r2',r4,r4'
ld   r3 = [r4]
ld   r3' = [r4']
add  r1 = r2, r3
add  r1' = r2', r3'
call addrCompare(r1, r1')
br   faultDet, r2 != r2'
st   [r1] = r2
st   [r1'] = r2'
```

CRAFT (Hybrid HW/SW)

```
// r2,r2',r4,r4'
dup.ld r3 = [r4]
       r3' = [r4']
add    r1 = r2, r3
add    r1' = r2', r3'
br     faultDet, r1 != r1'
br     faultDet, r2 != r2'
dup.st [r1] = r2
       [r1'] = r2'
```

SWIFT

```
// r2,r2',r4,r4'
br  faultDet, r4 != r4'
ld  r3 = [r4]
mov r3' = r3
add r1 = r2, r3
add r1' = r2', r3'
br  faultDet, r1 != r1'
br  faultDet, r2 != r2'
st  [r1] = r2
```

SWIFT-R

```
// r2,r2',r2",r4,r4',r4"
call voter(r4, r4', r4")
ld   r3 = [r4]
mov  r3' = r3
mov  r3" = r3
add  r1 = r2, r3
add  r1' = r2', r3'
add  r1" = r2", r3"
call voter(r1, r1', r1")
call voter(r2, r2', r2")
st   [r1] = r2
```

TRUMP

```
// r2,r2',r4,r4'
br  recover, 3*r4 != r4'
ld  r3 = [r4]
mul r3' = 3, r3
add r1 = r2, r3
add r1' = r2', r3'
br  recover, 3*r1 != r1'
br  recover, 3*r2 != r2'
st  [r1] = r2
```

Programmausschnitt 2.3: Softwarebasierte Fehlererkennung durch Replikation auf Instruktionsebene.
Vergleich verschiedener Techniken am Beispiel einer Adressabstandsberechnung (Register in den Kommentaren sind jeweils vordefiniert). Die Replikation des Berechnungspfades ermöglicht Fehlererkennung (EDDI, SWIFT, CRAFT) beziehungsweise Fehlertoleranz (SWIFT-R, TRUMP). Hierfür nutzt SWIFT-R ein zweites Replikat, während TRUMP Prüfsummen (Multiplikation mit 3) in ein Replikat einfügt. Die Ansätze unterscheiden sich weiterhin im erzielbaren Redundanzbereich: EDDI und CRAFT schließen beispielsweise Speicher und Adressierung mit ein, wobei CRAFT ein hybrider Ansatz und entsprechend auf spezielle Hardwareerweiterungen (`dup.ld` und `dup.st`) angewiesen ist. *Beispiele aus bzw. nach: [OSM02b, RCV+05a, CRA06]*

Die Diskussion der folgenden compilerbasierten Ansätze orientiert sich beispielhaft an der in Programmausschnitt 2.3 dargestellten Berechnung eines Adressabstands. Im Gegensatz zu dem ungeschützten Original werden die Verarbeitungsströme durch die Replikationsansätze in unterschiedlicher Weise aufgeteilt. Die Techniken unterscheiden sich hierbei im Wesentlichen durch die Art der Aufteilung und Zusammenführung der Ströme (`ld` und `st`). Die eingewobenen Funktionen (`faultDet` bzw. `voter`) dienen dem Vergleich der verschiedenen Verarbeitungsströme. Prinzipiell identisch ist hingegen die eigentliche Adressberechnung (`add`) im Mittelteil.

EDDI [OSM02b] dupliziert sowohl die Ausführung als auch die Daten (Register und Speicher). Aus diesem Grund sind spezielle Umsetzungsfunktionen (`addrCompare`) für die Adressen zu den replizierten Daten notwendig. Die Fehlererkennung beruht auf dem Vergleich der redundanten Daten vor der Speicherung. Die erzielbare Fehlererfassung ist lückenhaft und umfasst im weitesten Sinne Operatorfehler. Da beide Verarbeitungsströme auf demselben Hardwarekontext ausgeführt werden, sind Kontrollflussfehler kaum erkennbar. Die Evaluation [OSM02b] zeigt entsprechend auch in erheblichem Umfang unerkannte Datenfehler. In vielerlei Hinsicht gleicht SWIFT [RCV+05b] EDDI, es dupliziert jedoch nur die Instruktionen und schließt den Speicher unter der Annahme entsprechender Hardwareredundanz aus dem Redundanzbereich aus. Hierdurch erübrigt sich eine aufwendige Adressumsetzung, allerdings müssen die Daten im Gegenzug während der Ausführung dupliziert werden. Auch SWIFT führt die Verarbeitungsströme auf demselben Hardwarekontext aus, nutzt jedoch eine auf dem Ansatz von Schuette und Shen [SS87] aufbauende signaturbasierte Kontrollflussüberwachung (nicht dargestellt). Insgesamt ist die Fehlererfassung jedoch lückenhaft [RCV+05b, Muk08]. Eine um Fehlerbehandlung erweiterte Version

von SWIFT stellt SWIFT-R [CRA06] dar. Es nutzt hierfür einen dritten Verarbeitungsstrom sowie Mehrheitsentscheider für das Laden und Speichern von Daten. TRUMP [CRA06] hingegen ersetzt den dritten Verarbeitungsstrom durch arithmetische Codierung, genauer einem AN-Code mit konstantem $A = 3$. Die Rückführung der Codierung erfolgt durch eine angepasste Vergleichsfunktion (`recover`). Beide Varianten verbessern die Fehlererkennung gegenüber SWIFT, zeigen jedoch weiterhin signifikante Einschränkungen in der Fehlererfassung wie im entstehenden Redundanzbereich.

Im Vergleich zu dem hybriden Ansatz von CRAFT [RCV+05a] zeigen sich die Schwierigkeiten der rein softwarebasierten Replikation. Obwohl CRAFT auf den Konzepten von SWIFT basiert, kann dieser hybride Ansatz die SDC-Fehlerwahrscheinlichkeit um zwei Drittel senken. CRAFT nutzt spezifische Hardwareerweiterungen für die Duplizierung und Überprüfung der Daten beim Laden (`dup.ld`) beziehungsweise Speichern (`dup.st`). Mittels dieser Operationen schwächt CRAFT das Fehlermodell ab und vermeidet die ungeschützte Datenreplikation beziehungsweise Speichertransfers auf Kosten zusätzlicher Hardwareredundanz.

Analog zu den vorangegangenen compilergestützten Verfahren auf der Befehlssatzebene, lässt sich die Transformation ebenso auf dem Programmquellcode [RRTV99, RRVT01, NSV04] durchführen. Die erzielbaren Fehlertoleranzeigenschaften sind dabei weitgehend identisch.

Replikation auf Instruktionsebene hat den Nachteil der fehlenden Isolation zwischen den replizierten Verarbeitungsströmen. Die Replikate teilen sich einen gemeinsamen Hardwarekontext und sind entsprechend anfällig für Kontrollflussfehler. Die Fehlererfassung ist daher grundsätzlich unvollständig. Der Redundanzbereich schließt weiterhin jede Art von Peripherie und asynchronen Ereignissen aus. Von Vorteil ist hingegen die feingranulare, selektive Anwendbarkeit auf beliebige Teile des Systems. Insgesamt zeigt sich die Replikation auf Instruktionsebene jedoch als wenig geeignet für eine Implementierung der angestrebten Fehlertoleranzeigenschaften.

2.3 Zusammenfassung

Für die Auswahl und die Beurteilung geeigneter Fehlertoleranzmaßnahmen ist zunächst ein tieferes Verständnis der Fehlersymptomatik und der Fehlerausbreitung unumgänglich. Transiente Fehler manifestieren sich initial in den kleinsten Strukturelementen der Software: den Maschinenanweisungen. Durch einen simplen Bitfehler in einem Register kann es durch *Fehlerfortpflanzung* schließlich zu einem beobachtbaren Versagen der sicherheitskritischen Anwendung kommen. Die hierbei entstehende *Fehlerkette* gliedert sich, bezogen auf die jeweilige Betrachtungsebene, in *Defekt*, *Fehler* und *Fehlverhalten*. Fehlertoleranzmaßnahmen sollen diese Fehlerkette gezielt unterbrechen und ein Fehlverhalten verhindern. Ihre Effektivität wird daher häufig gemäß der Fehlerraten für *erkannte nicht behebbare Fehler* (DUE) und *unerkannte Datenfehler* (SDC) klassifiziert. Letztere sind in diesem Zusammenhang ein wichtiger Indikator für *kritische Fehlerstellen* und fehlende Redundanz in den entsprechenden Fehlertoleranzmaßnahmen. Ob ein Fehler unentdeckt bleibt, hängt auch davon ab, inwieweit die Fehlererkennung in der Lage ist, das *Fehlermodell* bestehend aus Operator, Operanden und Berechnungsfehlern abzudecken. Der koordinierte Einsatz von Redundanz ermöglicht schließlich die Implementierung geeigneter Fehlertoleranz. Hierfür stehen verschiedene Entwurfsalternativen zur Verfügung. Dies sind zunächst die vier grundlegenden *Redundanzarten*: Information, Zeit, Funktion und Struktur. Für deren Einsatz existieren verschiedene, darauf aufbauende *Redundanzstrategien*. Eine Strategie, wie beispielsweise die Codierung, ist eine Blaupause für die konkrete Implementierung einer spezifischen *Redundanztechnik* auf einer bestimmten Ebene des Systems. Der *Redundanzbereich* beschreibt den räumlichen wie zeitlichen Wirkungsbereich der entsprechenden Technik und erlaubt die Lokalisierung kritischer Fehlerstellen und ungeschützter Bereiche. Die Fehlererkennung alleine reicht für ein fehlertolerantes Verhalten

jedoch nicht aus. Dieses entsteht erst durch die geeignete *Fehlerdiagnose* und *Fehlerbehandlung*. Die Fehlerdiagnose kann dabei in Abhängigkeit von der gewählten Strategie durch *Absolut-* oder *Relativtests* erfolgen. Ist der Fehlerbereich identifiziert, kann der Fehler durch *Vorwärtskorrektur*, *Rückwärtskorrektur* oder *Maskierung* behandelt werden. Die dreifache Redundanz (TMR) ist ein Beispiel für die Fehlerdiagnose durch einen Relativtest sowie die Maskierung des Fehlers durch Ausschluss des entsprechenden Replikats.

Mit Blick auf den verfolgten Lösungsansatz scheiden Fehlertoleranzmaßnahmen und Redundanztechniken aus, die sich ausschließlich mithilfe von Anwendungswissen beziehungsweise durch funktionale Änderungen an der Anwendung umsetzen lassen. Hierzu zählt insbesondere der Einsatz von funktioneller oder zeitlicher Redundanz zur Fehlererkennung. Vielversprechend sind hingegen Ansätze, die auf Informationsredundanz beziehungsweise struktureller Redundanz basieren.

Eine allgemeine Strategie zur Nutzung von Informationsredundanz ist die *Codierung* von Daten. Über die reine Datencodierung spielen vor allem Ansätze der *arithmetischen Codierung* eine entscheidende Rolle, da sie Fehler in der eigentlichen Programmausführung aufdecken können. Von den verschiedenen Strategien bieten lediglich die ANBD-Codes eine vollständige Fehlererfassung im Sinne des Fehlermodells. Der Aufbau eines geschlossenen Redundanzbereiches erfordert jedoch in jedem Fall die lückenlose Codierung der gesamten Programmausführung. Konzeptbedingt ist arithmetische Codierung jedoch auf ganzzahlige Arithmetik beschränkt. Zudem ist der Umfang der zur Verfügung stehenden Operatoren eingeschränkt und schließt beispielsweise die Division üblicherweise aus. Durch die notwendige Nachbildung komplexerer Zahlendarstellungen beziehungsweise arithmetischer Operatoren sind die induzierten Laufzeitkosten erheblich. Da fehlerkorrigierende Verfahren für die Umsetzung in Software nicht zur Verfügung stehen, sind zusätzliche Maßnahmen für die Implementierung eines fehlertoleranten Verhaltens erforderlich.

Demgegenüber basiert *Replikation* auf der redundanten Ausführung von Strukturelementen. Die Anzahl der Replikate bestimmt dabei, ob lediglich Fehlererkennung oder Fehlerbehandlung durch Maskierung möglich ist. Die Fehlererfassung durch den Relativtest der Ergebnisse ist vollständig hinsichtlich des Fehlermodells, sofern die Fehlerausbreitung zwischen den Replikaten ausgeschlossen ist. Von entscheidender Bedeutung für die Effektivität der Replikation ist dementsprechend die Isolation der redundanten Verarbeitungsströme sowie die geeignete Vervielfältigung der Eingabedaten auf die Replikate. Weiterhin stellt der Mehrheitsentscheider am Ende der replizierten Ausführung eine kritische Fehlerstelle dar. Einen maßgeblichen Einfluss auf diese Faktoren hat die gewählte Implementierungsebene. Auf der einen Seite steht die Replikation des Gesamtsystems mittels virtueller Maschinen (RVM). Der umfassende Redundanzbereich sowie die gute Isolation der Replikate geht hier auf Kosten der Selektivität. Auf der anderen Seite lässt die selektive Replikation auf der Instruktionsebene in hohem Maße die Fehlerausbreitung zwischen den replizierten Verarbeitungsströmen zu. Ansätze auf dieser Ebene weisen entsprechend signifikante SDC-Fehlerraten auf. Einen Mittelweg eröffnet die Replikation von Prozessinkarnationen. Die Abbildung der Anwendung auf Prozesse des Betriebssystems stellt einen integralen Bestandteil des regulären Systementwurfs dar. Daher ist die Abgrenzung der zu replizierenden Elemente üblicherweise relativ einfach. Erforderlich sind in diesem Fall hingegen geeignete Maßnahmen zur Sicherstellung des Replikdeterminismus sowie der Zuverlässigkeit der Mehrheitsentscheider.

„Sed quis custodiet ipsos custodes?“
„Wer aber wird die Wächter selbst bewachen?“
— Juvenal*, Satire 6, 347 f.*

3

Der Kombinierte Redundanzansatz

Combined Redundancy (CoRed)

Im vorangegangenen Kapitel wurden die verschiedenen Aspekte der softwarebasierten Fehlertoleranz anschaulich aufgezeigt. Arithmetische Codierung nutzt beispielsweise Informationsredundanz für die Fehlererkennung. Die Forschung in diesem Bereich zeichnet sich vor allem durch die Verbesserung der erzielbaren Fehlererfassung aus. War diese bei den AN-Codierung noch unvollständig, so lassen sich mittels ANBD-Codes konzeptionell alle in Software sichtbaren Fehler erkennen. In der praktischen Umsetzbarkeit leidet die arithmetische Codierung jedoch an den hohen Laufzeitkosten und einer für die codierte Ausführung der gesamten Anwendung notwendigen, technisch komplexen Infrastruktur. Letztere stellt eine potenzielle Schwachstelle dar und führt bei den bekannten Ansätzen dennoch in gewissem Umfang zu unerkannten Datenfehlern (vgl. Kapitel 2.2.2). Der Einsatz struktureller Redundanz durch Replikation weist grundsätzlich ebenfalls eine vollständige Fehlererfassung im Zuge der redundanten Ausführung auf. Die verwandten Arbeiten konzentrieren sich hier vorrangig auf die verschiedenen Implementierungsebenen sowie auf die Einsparung beziehungsweise Substitution von Replikaten zur Optimierung der Laufzeitkosten. Die Ansätze unterscheiden sich im Einzelnen vor allem im Umfang der Replikation und damit im erzielbaren Redundanzbereich. Dieser weist jedoch in jedem Fall Lücken durch die unvermeidliche Replikation der Eingabedaten sowie die Mehrheitsentscheider auf. In diesen unreplizierten Bereichen treten wiederum unerkannte Datenfehler in signifikantem Umfang auf [AVFK02].

Kritische Fehlerstellen und fehlende Redundanz schwächen die Effektivität der eingesetzten Fehlertoleranz und erschweren in der Folge auch die praktische Umsetzbarkeit. Die bestehenden softwarebasierten Fehlertoleranzansätze erweisen sich entsprechend als unzureichend für die Umsetzung einer vollständigen Fehlererkennung und gleichzeitig eines durchgehenden Redundanzbereichs, wie sie für eine gezielte Beeinflussung der Zuverlässigkeit wünschenswert ist. Es fehlt ein ganzheitlicher Fehlertoleranzansatz, der sowohl eine vollständige Fehlererfassung als auch einen in sich geschlossenen Redundanzbereich ermöglicht, ohne dabei die Selektivität hinsichtlich des Schutzes einzelner sicherheitskritischer Anwendungen zu verlieren.

Mit Blick auf die Ziele dieser Arbeit stellt die Replikation auf Betriebssystemebene einen geeigneten Ausgangspunkt dar. Zum einen sind Prozessinkarnationen ein gängiges Strukturelement

für die logische Aufteilung und Abbildung der Anwendung auf das Betriebssystem. Ausgehend von der hierdurch bereits vorgegebenen Struktur lässt sich die Replikation somit vergleichsweise einfach durch eine Vervielfältigung der Prozessinkarnationen der Anwendung erreichen. Sie lässt sich damit weitgehend transparent in den Entwicklungsprozess einfügen und ist dennoch selektiv auf die sicherheitskritischen Anwendungen in einem System anwendbar. Zum anderen erfordert die redundante Ausführung aufgrund des einfachen Relativtests im Allgemeinen kein spezifisches Anwendungswissen und ermöglicht dennoch eine vollständige Fehlerauffassung im Sinne des Fehlermodells (vgl. Kapitel 2.1.1).

Auf der Basis der softwarebasierten Replikation entwickelt dieses Kapitel den ***kombinierten Redundanzansatz* CoRed** (engl. *Combined Redundancy*) und damit das Rüstzeug für die gezielte Beeinflussung der nicht-funktionalen Eigenschaft Zuverlässigkeit auf Ebene des Betriebssystems. Zur genaueren Charakterisierung der kritischen Fehlerstellen in der redundanten Ausführung erfolgt zunächst eine erweiterte Problemanalyse in Kapitel 3.1 sowie eine Spezifikation der dem Ansatz zugrunde liegenden Annahmen und der Fehlerhypothese in Kapitel 3.2. Darauf aufbauend gibt Kapitel 3.3 zunächst einen groben Überblick über das Architekturkonzept und die allgemeine Methodik des CoRed-Ansatzes. Die dort skizzierten Maßnahmen führen anschließend zu den zentralen Erweiterungen der softwarebasierten Replikation in Form von Schutzelementen für die Eingänge und Ausgänge der redundanten Ausführung in Kapitel 3.4 und zuverlässiger Mehrheitsentscheider im darauffolgenden Kapitel 3.5. Zusammengenommen stellt CoRed einen holistischen Fehlertoleranzansatz für die selektive Absicherung sicherheitskritischer Anwendungen dar. Der Ansatz erlaubt dabei zugleich den Aufbau eines geschlossenen Redundanzbereichs sowie eine vollständige Fehlererfassung von in Software sichtbaren transienten Fehlern.

3.1 Erweiterte Problemanalyse

Die Fehlerrate für transiente Fehler ist eine inhärente Eigenschaft der Hardware (vgl. Kapitel 1.1). Entsprechend erfordert die Steigerung der Zuverlässigkeit die Senkung der sichtbaren *effektiven Fehlerrate*. Die von der Hardware ausgehenden Fehlerkette (vgl. Kapitel 2.1.1) muss daher durch softwarebasierte Fehlertoleranzmaßnahmen zuverlässig durchbrochen werden, bevor es zu einem Fehlverhalten der sicherheitskritischen Anwendung kommen kann.

Den Ausgangspunkt hierfür stellt die Replikation der Anwendung auf Betriebssystemebene in Form dreifacher Redundanz (TMR) dar. Dieses bekannte und erprobte Verfahren bietet eine effektive Fehlererkennung im Rahmen der replizierten Ausführung. Gleichwohl lässt sich die Implementierung und Überwachung der Replikation selbst nicht redundant auslegen. Treten Fehler in diesen unreplizierten Bereichen auf, ist eine Unterbrechung der Fehlerkette nicht mehr gewährleistet und es kann weiterhin zu einem Fehlverhalten der sicherheitskritischen Anwendung kommen. Diesen Zusammenhang konnten Aidenmark et al. [AVFK02] anschaulich am Beispiel einer replizierten Brake-by-Wire Anwendung zeigen. In ihrer Evaluation ordnen sie die verbleibende Fehlerrate unter anderem auch den kritischen Fehlerstellen in der Replikation zu. Generell kann ein eingeschränkter Redundanzbereich zu einer dramatischen Verschlechterung der Effektivität der eingesetzten Fehlertoleranz führen [Arn73, AVFK02]. Bouricius et al. [BCS69] stufen die Abdeckung der Replikation sogar als *den* wichtigsten Entwurfsparameter ein. Bereits eine vergleichsweise kleine Reduktion der Abdeckung kann demnach zu einer Verschlechterung der Zuverlässigkeit um Größenordnungen führen. Die folgende erweiterte Problemanalyse dient daher der Identifikation von kritischen Fehlerstellen und Lücken im Redundanzbereich der Replikation.

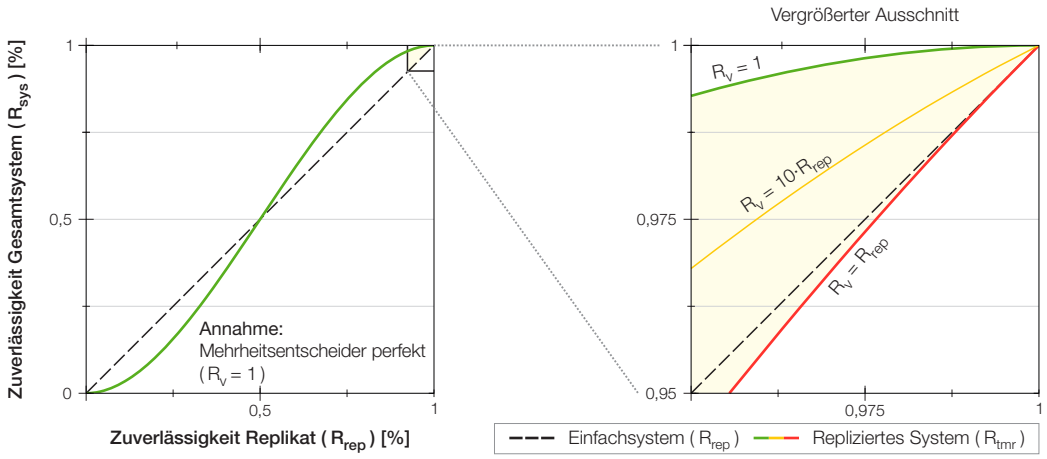


Abbildung 3.1: Zuverlässigkeit eines dreifach redundanten Systems.
Die Gesamtzuverlässigkeit R_{sys} bestimmt sich aus der Zuverlässigkeit der Replikate R_{rep} sowie der des Mehrheitsentscheiders (engl. *voter*) R_v. Die Gesamtzuverlässigkeit des 3-fachen Systems übertrifft die des Einfachsystems erst ab $R_{rep} > 0{,}5$ und unter der Annahme eines perfekten Mehrheitsentscheiders (links). Hingegen verschlechtert sich R_{sys} dramatisch mit der sinkenden Zuverlässigkeit des Mehrheitsentscheiders (rechts). In diesem Beispiel ist der Mehrheitsentscheider zehnmal zuverlässiger als die Replikate. Entsprechen sich R_v und R_{rep}, ist das 3-fach System in jedem Fall unzuverlässiger als das Einfachsystem.

3.1.1 Kritische Fehlerstellen in der Replikation

Im Allgemeinen wird die Überlebenswahrscheinlichkeit eines Systems mittels der Zuverlässigkeitsfunktion $R(t)$ angegeben. Diese dient im Folgenden als illustratives Maß für den Einfluss der einzelnen Bausteine der Replikation, wobei aus Gründen der Übersichtlichkeit die vereinfachte Darstellung R als die angenommene Zuverlässigkeit für einen Betriebszyklus der sicherheitskritischen Anwendung verwendet wird.

Abbildung 3.1 (links) veranschaulicht die Auswirkungen der Replikation auf die Gesamtzuverlässigkeit (R_{sys}) eines Systems. Diese entspricht in einem Einfachsystem der Zuverlässigkeit eines Replikats ($R_{sys} = R_{rep}$). Im replizierten Fall (R_{tmr}) setzt sich die Gesamtzuverlässigkeit zusammen aus der Zuverlässigkeit des Mehrheitsentscheiders (R_v) sowie der Wahrscheinlichkeit für die korrekte Ausführung von mindestens zwei der drei Replikate [LV62]:

$$R_{tmr} = R_v \cdot \underbrace{(3R_{rep}^2 - 2R_{rep}^3)}_{\text{2-aus-3 Replikate}} \tag{3.1}$$

Da die zusätzlichen Replikate ihrerseits eine potenzielle Fehlerquelle darstellen, übertrifft die Zuverlässigkeit des replizierten Systems die des Einfachsystems erst ab einer Replikatzuverlässigkeit von $R_{rep} > 0{,}5$ (Schnittpunkt in Abbildung 3.1, links). Aufgrund der relativ geringen Fehlerraten transienter Fehler kann von einer für die Replikation sinnvollen Replikatzuverlässigkeit von $R_{rep} \gg 0{,}9$ ausgegangen werden. Eine dieser Betrachtung zugrunde liegende Annahme ist jedoch die Unfehlbarkeit des Mehrheitsentscheiders ($R_v = 1$). Diese als inhärent fehlerfrei angenommenen Teile des Systems werden als *Perfektionskerne*[1] (engl. *hardcore*) [Ech90] bezeichnet (siehe auch Abbildung 3.2). Bei der physikalischen Replikation der Hardware wird diese Eigenschaft näherungsweise durch hochzuverlässige Schaltungen für den Vergleich der Ausgänge erreicht.

[1] Aus der Systemsicherheit (engl. *security*) entliehen, wird vor allem bei der softwarebasierten Redundanz auch der Begriff *Trusted Computing Base* (TCB) verwendet.

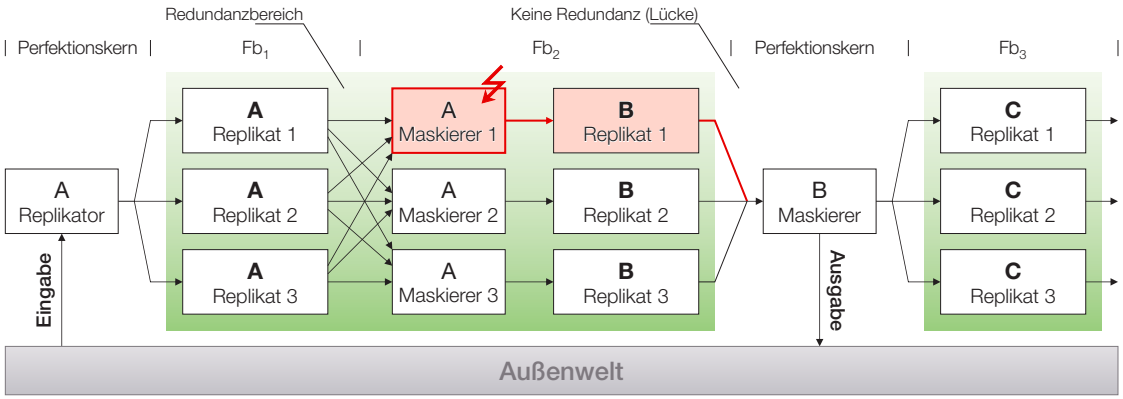


Abbildung 3.2: Entwurfsalternativen eines dreifach redundanten Systems.
Dreifache Redundanz nimmt üblicherweise sowohl die Replikation der Eingaben wie auch den Mehrheitsentscheid als fehlerfrei an. Diese Komponenten liegen entsprechend in einem *Perfektionskern* außerhalb der Fehlerbetrachtung. Ist diese Unfehlbarkeit nicht gegeben, begrenzen diese kritischen Fehlerstellen die Gesamtzuverlässigkeit. Eine Alternative stellen *redundante Mehrheitsentscheider* (Maskierer) dar. Diese vermeiden zwar den Zuverlässigkeitsengpass, lassen sich jedoch nur zwischen Replikationsdomänen sinnvoll nutzen. Durch Fehlerausbreitung auf das zugeordnete Replikat (hier B_1) kann ein tatsächliches Fehlerverhalten eines verbleibenden Replikats (hier B_2 oder B_3) nicht toleriert werden. Maskierer und Replikat versagen folglich nicht mehr unabhängig voneinander und werden daher demselben *Fehlerbereich* (Fb_2) zugeordnet.

Der Einfluss der Mehrheitsentscheider

Die Ausführung der Replikate und Mehrheitsentscheider erfolgt bei der softwarebasierten Replikation jedoch auf derselben unzuverlässigen Hardware. Da die Gesamtzuverlässigkeit durch den nur einmal vorhandenen Mehrheitsentscheider dominiert und begrenzt wird (vgl. Gleichung 3.1), sinkt der mögliche Zugewinn durch die dreifache Redundanz dramatisch mit der Fehlerwahrscheinlichkeit des Mehrheitsentscheiders. Die Auswirkungen zeigt Abbildung 3.1 (rechts) anhand eines vergrößerten Ausschnitts der Systemzuverlässigkeit. Selbst bei einer, im Vergleich zu den Replikaten, zehnfach höheren Zuverlässigkeit des Mehrheitsentscheiders, sind die Auswirkungen auf die Gesamtzuverlässigkeit signifikant. Diese kann zudem konzeptbedingt keinesfalls oberhalb der Zuverlässigkeit des Mehrheitsentscheiders liegen, womit dieser eine schwerwiegende kritische Fehlerstelle in der redundanten Ausführung darstellt.

Redundante Mehrheitsentscheider

Eine Möglichkeit diesen Zuverlässigkeitsengpass zu vermeiden, besteht in der redundanten Auslegung des Mehrheitsentscheids selbst. Der Aufbau eines solchen *redundanten Mehrheitsentscheiders* ist in Abbildung 3.2 zwischen den Replikatblöcken A und B dargestellt. Jedes Replikat überträgt sein Ergebnis an jeden Mehrheitsentscheider, deren Ausgänge wiederum als Eingabe für den nächsten Block dienen. Auf diese Weise lassen sich sowohl der Mehrheitsentscheider als kritische Fehlerstelle als auch die Replikation der Eingabedaten für den nachfolgenden Block B eliminieren. Die Gesamtzuverlässigkeit ($R_{m\text{-}tmr}$) unter Einsatz des redundanten Mehrheitsentscheiders ist nunmehr definiert als [Lal00]:

$$R_{m-tmr} = R_v^2 \cdot (\underbrace{3R_{rep}^2 - 2R_{rep}^3}_{\text{2-aus-3 Replikate}} R_v) \tag{3.2}$$

Auf den ersten Blick löst dieses Konzept das Problem des Mehrheitsentscheiders als kritische Fehlerstelle, da ein Ausfall eines Maskierers nicht mehr zwangsläufig zu einem Ausfall des Gesamtsystems führt. Gleichzeitig stellt die Zuverlässigkeit des Mehrheitsentscheiders keinen limitierenden Faktor mehr für die Gesamtzuverlässigkeit dar.

Demgegenüber ergeben sich jedoch auch eine Reihe von Nachteilen und Einschränkungen aus dem redundanten Aufbau. Da die Ausgabe wiederum aus drei Ergebnissen besteht, bleibt der eigentliche Sinn und Zweck der Mehrheitsentscheidung augenscheinlich unerfüllt. Entsprechend lassen sich redundante Mehrheitsentscheider lediglich zwischen einzelnen TMR-Stufen sinnvoll nutzen. Eine Ausgabe der Ergebnisse aus der replizierten Domäne an die Außenwelt ist nicht möglich, ebenso wenig wie die Interaktion mit der Systemsoftware. Insofern beschränkt sich die Einsatzmöglichkeit dieses Ansatzes auf die Verkleinerung der Replikate, beispielsweise zur Verringerung der Fehlererkennungslatenzen. Weiterhin nimmt der Einfluss des Mehrheitsentscheiders auf die Gesamtzuverlässigkeit zwar ab, ein signifikanter und nicht zu vernachlässigender Zusammenhang (vgl. Gleichung 3.2) besteht jedoch nach wie vor.

Ein weiteres Problem ergibt sich aus der möglichen Fehlerfortpflanzung zwischen Mehrheitsentscheider und Replikat. Im Beispiel in Abbildung 3.2 pflanzt sich das Fehlverhalten des Maskierers A_1 auf das Replikat B_1 fort, dessen Fehlverhalten wiederum erst durch den darauffolgenden Mehrheitsentscheid aufgedeckt wird. Ein üblicherweise tolerierbares Fehlverhalten eines der Replikate, in diesem Beispiel B_2 oder B_3, führt damit zu einem Versagen der Fehlertoleranz insgesamt. Der redundante Maskierer und das zugeordnete Replikat versagen nicht mehr unabhängig voneinander und bilden folglich einen gemeinsamen *Fehlerbereich* (Fb_2) [Ech90]. Als Gegenstück zum Redundanzbereich charakterisiert der Fehlerbereich das *angenommene* zeitliche wie räumliche Auftreten von Fehlern. Die im Sinne der Fehlertoleranz voneinander unabhängigen Komponenten bilden die sogenannten *Einzelfehlerbereiche*. Im Beispiel in Abbildung 3.2 entsprechen die Replikate der Prozessinkarnationen A und C jeweils einem Einzelfehlerbereich (Fb_1 und Fb_3). Demgegenüber ist eine unabhängige Fehlerbetrachtung bei der Replikatgruppe B nicht möglich. Hier müssen Maskierer und Replikat zu einem Bereich (Fb_2) zusammengefasst werden.

Zuverlässigkeitsengpass Mehrheitsentscheider

Der Ausfall eines Mehrheitsentscheiders ist grundsätzlich nicht tolerierbar, seine Zuverlässigkeit dominiert daher die Gesamtzuverlässigkeit der redundanten Ausführung. Der Mehrheitsentscheider stellt somit eine schwerwiegende kritische Fehlerstelle dar, welche potenziell die praktische Umsetzbarkeit softwarebasierter Redundanz gefährdet.

Die Mehrheitsentscheider stellen insgesamt einen ernst zu nehmenden Zuverlässigkeitsengpass in der redundanten Ausführung dar, da sie unabhängig von den Replikaten versagen können und entsprechend ihrerseits eigenständige Einzelfehlerbereiche darstellen. Ihre Anzahl ist in hohem Maße anwendungsabhängig, da jeder Interaktionspunkt mit der Umgebung die Herbeiführung einer eindeutigen Ausgabe erfordert. Einfache Mehrheitsentscheider stellen in jedem Fall eine kritische Fehlerstelle dar, deren Versagen nicht tolerierbar ist. Demgegenüber bieten redundante Mehrheitsentscheider den Schutz der Replikation. Sie sind jedoch lediglich für eine Verkleinerung der Replikatgröße sinnvoll einsetzbar, da sie das grundlegende Problem der konsistenten Ausgabe nicht lösen. Unabhängig von der konkreten Umsetzung, wirkt sich eine Unzuverlässigkeit der Mehrheitsentscheider negativ auf die Gesamtzuverlässigkeit aus (vgl. Gleichung 3.1 und Gleichung 3.2). Da sich das Problem der Mehrheitsfindung und Ausgabe mittels redundanter Maskierer nicht lösen lässt, bleibt letztlich nur die Verbesserung der Zuverlässigkeit des einfachen Mehrheitsentscheiders.

Fehlertoleranz von Eingang zu Ausgang

Neben den Mehrheitsentscheidern existieren jedoch noch weitere Teile, die sich nicht replizieren lassen. Die entstehenden Lücken im Redundanzbereich werden hierbei im Wesentlichen durch Datenflüsse innerhalb der replizierten Domäne beziehungsweise durch Interaktionspunkte mit der Außenwelt verursacht.

Die initiale Replikation der Eingaben markiert den Startpunkt des Datenflusses durch die redundante Ausführung. Im Beispiel in Abbildung 3.2 ist dies der Replikator A (engl. *replicator*). Da die Eingaben bei der asynchronen Replikation durch den zeitlichen Versatz voneinander abweichen können, agiert der Replikator als *Stellvertreter* (engl. *proxy*) gegenüber der Außenwelt und vervielfältigt die Eingaben auf die Replikate. Im weiteren Verlauf der redundanten Ausführung erfolgt die Replikation der Ergebnisse auf die Replikate implizit durch den Maskierer, beispielsweise zwischen den Blöcken B und C. Da ein Versagen der Eingabereplikation im Allgemeinen nicht tolerierbar ist, wird der Replikator wie schon der Mehrheitsentscheider üblicherweise dem Perfektionskern zugeordnet. Analog zu den Auswirkungen bei den Mehrheitsentscheidern, dominiert und begrenzt die Zuverlässigkeit der Eingabereplikation die Gesamtzuverlässigkeit gemäß Gleichung 3.1.

Grenzen der Replikation

Die initiale Replikation der Eingaben sowie die finale Ausgabe der Ergebnisse bilden die natürlichen Grenzen der Replikation. Beide Endpunkte liegen jedoch bereits außerhalb des Redundanzbereichs. Unabhängig davon stellt die Eingabereplikation innerhalb der redundanten Ausführung in jedem Fall eine kritische Fehlerstelle dar. Die Auswirkungen auf die Gesamtzuverlässigkeit entsprechen denen des Mehrheitsentscheiders.

Innerhalb der replizierten Ausführung stellt die Replikation der Eingabedaten einen mit den Mehrheitsentscheidern vergleichbaren Zuverlässigkeitsengpass dar. Wiederum lässt sich das Problem durch weitere Replikationsstufen nicht lösen und es bleibt nur die Verbesserung der Zuverlässigkeit der entsprechenden Komponenten. Darüber hinaus wird die Replikation zwar durch die initiale Eingabe sowie die finale Ausgabe in natürlicher Weise begrenzt, allerdings beginnt der Redundanzbereich erst mit den Replikaten. Der Eingangsreplikator wie auch die Übermittlung der Ausgaben liegen außerhalb und stellen ihrerseits kritische Fehlerstellen dar. Eine nahtlose Verknüpfung verschiedener Anwendungen und Replikationsdomänen ist so nicht möglich [Sch90].

3.1.2 Fehlererfassung und Zuverlässigkeit

Wie bereits erwähnt, bietet die Replikation eine vollständige Fehlererfassung. Da die Art und Schwere der auftretenden Fehler dabei im Wesentlichen unerheblich ist, lässt sich die Zuverlässigkeit der Replikate relativ einfach aus der Fehlerrate der Hardware ableiten [Muk08]. Die Risikoabschätzung kann in der Folge gemäß der zuvor eingeführten Zuverlässigkeitsfunktion R_{tmr} erfolgen. Für die praktische Umsetzbarkeit softwarebasierter Replikation steht daher insbesondere die Vermeidung der Fehlerausbreitung und die Unabhängigkeit der Fehlerbereiche im Vordergrund. Kritische Fehlerstellen unterlaufen und korrumpieren diese Eigenschaften, weshalb sie explizit in die Risikoabschätzung einfließen müssen [Ech90].

Um dieses Problem zu lösen, ordnen die in Kapitel 2.2.3 diskutierten verwandten Arbeiten den Mehrheitsentscheider sowie die Eingangsreplikation im Allgemeinen dem Perfektionskern zu. Eine übliche Grundannahme ist weiterhin die räumliche wie zeitliche Gleichverteilung der auftretenden transienten Fehler, welche es erlaubt, die Fehlerwahrscheinlichkeit und damit die Auswirkungen auf die Zuverlässigkeit in Relation zur Ausführungszeit zu setzen [GRRV06b]. Die Ermittlung

der realen Ausfallwahrscheinlichkeit kritischer Fehlerstellen ist jedoch nicht trivial und hängt unter anderem auch von den individuellen Fehlerraten in den tatsächlich genutzten Teilen der Hardware ab [MWE+03, WQRP04, MER05, RCBS07]. Weiterhin legen aktuelle Studien [NDO11] den Schluss nahe, dass die angenommene Gleichverteilung der Fehler in der Praxis nicht gegeben ist. Insgesamt erscheint die Grundannahme einer kurzen und daher weitgehend zuverlässigen Funktion von Mehrheitsentscheider und Replikator fraglich.

Eine weitere Herausforderung für die Abschätzung der Zuverlässigkeit ergibt sich aus der Zahl der kritischen Fehlerstellen. Ihr Umfang ist bei der hardwarebasierten Replikation durch den Aufbau der Hardware unveränderlich vorgegeben. Demgegenüber ergeben sich diese Eigenschaften bei der softwarebasierten Replikation aus der sicherheitskritischen Anwendung selbst. Die Zusammensetzung der kritischen Fehlerstellen ändert sich demnach potenziell mit jeder Anpassung der Anwendung. In der Summe wird die Risikoabschätzung durch die unsichere Fehlerverteilung und die Anwendungsabhängigkeit in jedem Fall erheblich erschwert. Die Eliminierung der kritischen Fehlerstellen wirkt diesem Problem durch eine Verringerung der gesondert zu betrachtenden Stellen entgehen. Insgesamt ist ein nach außen hin fehlerfreies Verhalten der Mehrheitsentscheider und Replikatoren im Sinne eines Perfektionskerns erstrebenswert.

3.2 Grundannahmen

Wie alle technischen Maßnahmen ist auch Fehlertoleranz gewissen Grenzen unterworfen. Deren Entwurf erfordert daher in jedem Fall Grundannahmen hinsichtlich der Systemeigenschaften sowie der Art und dem Umfang der zu tolerierenden Fehler. Um die geforderten Eigenschaften erfüllen zu können, muss und kann Fehlertoleranz letztlich von einer bestimmten Anzahl an fehlerfreien Komponenten ausgehen [Ech90].

3.2.1 Fehlerhypothese und Fehlerbereichsannahme

Ganz allgemein richtet sich die Anzahl (n) der für die Replikation notwendigen Knoten nach der Zahl der zu tolerierenden Fehler (f) sowie der *Art des Fehlverhaltens*. Kopetz [Kop97] teilt diese in die folgenden drei Kategorien ein, wobei jeweils die Konsensbildung durch die Replikate selbst mit eingeschlossen ist:

Inaktiv bei Ausfall (engl. *fail-silent*)**:** Das Replikat liefert *korrekte* oder *keine* Ergebnisse. Ein Fehlverhalten führt entsprechend zur Einstellung der normalen Funktion des Replikats und ist folglich auch als solches erkennbar.
$\rightarrow$ Anzahl der erforderlichen Replikate: $n = f + 1$

Widerspruchsfrei bei Ausfall (engl. *fail-consistent*)**:** Das Replikat setzt seine Funktion im Fehlerfall unter Umständen fort und liefert *fehlerhafte* Ergebnisse. Diese treten als unerkannte Datenfehler nach außen in Erscheinung, es herrscht jedoch eine konsistente Sicht auf das Fehlverhalten.
$\rightarrow$ Anzahl der erforderlichen Replikate: $n = 2 \cdot f + 1$

Bösartig (engl. *malicious*)**:** Im Fehlerfall kann es wiederum zu unerkannten Datenfehlern kommen, das Replikat produziert darüber hinaus jedoch *verschiedene* Ergebnisse. Es herrscht zudem *keine* konsistente Sicht auf das Fehlverhalten. Diese Art des Fehlverhaltens steht häufig synonym für die sogenannten *byzantinischen Fehler* (engl. *byzantine failures*) [LSP82].
$\rightarrow$ Anzahl der erforderlichen Replikate: $n = 3 \cdot f + 1$

Abgrenzung zur byzantinischen Fehlertoleranz in verteilten Systemen

Byzantinische Fehlertoleranz (engl. *byzantine fault tolerance*) ist vorrangig in verteilten Systemen mit asynchroner Kommunikationsinfrastruktur relevant. Die einzelnen Knoten arbeiten hier unabhängig voneinander und es existiert per se keine konsistente Sicht auf das jeweilige Einzelverhalten beziehungsweise den Replikatzustand. In der Folge ist keine Unterscheidung zwischen Nachrichtenausfall und Fehlverhalten möglich. Kann ein bösartiges Fehlverhalten einzelner Knoten nicht ausgeschlossen werden, benötigt die zuverlässige Konsensbildung in der Folge $3 \cdot f + 1$ Replikate.

Demgegenüber ist die Bereitstellung einer konsistenten Sicht auf die Replikatzustände innerhalb eines Rechensystems im Allgemeinen einfach realisierbar. Die Replikate stellen ihre Ergebnisse synchron zur Verfügung wodurch sich ein Fehlerverhalten eindeutig von einem Ausfall unterscheiden lässt. Auf eine aufwendige Konsensbildung unter der Annahme eines Nachrichtenausfalls kann daher verzichtet werden. Insgesamt erlaubt die von der eigentlichen Replikatausführung abgekoppelte Konsensbildung durch den Mehrheitsentscheider die Erkennung *bösartiger* Fehler mittels $2 \cdot f + 1$ Replikaten [CNV04].

Fehlerbereichsannahme

Die dem CORED-Ansatz zugrunde liegende Fehlerhypothese geht weiterhin von der *Einzelfehlerannahme* (engl. *single-error assumption*) aus, wie sie typischerweise für die dreifache Redundanz angenommen wird [VN56]. Folglich soll zu jedem beliebigen Betrachtungszeitpunkt höchstens *ein* Einzelfehlerbereich fehlerhaft sein. Anschaulich entspricht dies einem Fehlverhalten von bis zu einem Replikat in jeder TMR-Stufe. Ebenso werden die kritischen Fehlerstellen nun als unabhängige Fehlerbereiche angesehen, welche ihrerseits durch transiente Fehler beeinträchtigt werden können. Somit soll auch in den Mehrheitsentscheidern und den Replikatoren mit dem Auftreten von jeweils einem Fehler gerechnet werden – die Replikationsinfrastruktur unterliegt dementsprechend derselben Fehlerannahme wie die redundante Ausführung selbst. In dem konkreten Beispiel einer mehrstufigen Replikation aus Abbildung 3.2 auf Seite 56 erlaubt diese Fehlerbereichsannahme *jeweils einen Fehler je Fehlerbereich* und demnach insgesamt bis zu fünf Einzelfehler.

3.2.2 Systemmodell

Zusätzlich zu der vorangegangenen Fehlerhypothese ergeben sich die folgenden Annahmen an die für die Ausführung notwendige Laufzeitumgebung sowie die zu schützende Anwendung.

Eigenschaften der Laufzeitumgebung

Ein Übergreifen eines Fehler beispielsweise auf andere Replikate kann zu einem gleichzeitigen Ausfall in mehreren Einzelfehlerbereichen und damit zu einer Verletzung der Fehlerhypothese führen. Um ein unabhängiges Versagen der Einzelfehlerbereiche sicherzustellen, sind derartige Gleichtaktfehler (engl. *common mode failures*) daher grundsätzlich zu vermeiden.

Die Laufzeitumgebung, bestehend aus Echtzeitbetriebssystem und der darunter liegenden Hardware, soll eine strikte *Fehlereingrenzung* (engl. *fault-containment*) erlauben. Hierfür sind die Einzelfehlerbereiche (Replikate, Mehrheitsentscheider, Replikatoren) in geeigneter Weise räumlich wie zeitlich gegeneinander abzuschotten. Die räumliche Isolation der partiellen Virtualisierung verhindert in diesem Zusammenhang die Korrumpierung des internen Zustands der Prozessinkarnationen untereinander. Darüber hinaus werden die verschiedenen Aktivitätsträger zeitlich voneinander isoliert, um beispielsweise eine Monopolisierung des Prozessors auszuschließen. Dies wird zum Beispiel durch eine strikt periodische Ausführung sowie einer entsprechenden Laufzeitkontrolle durch das Echtzeitbetriebssystem ermöglicht.

Für die Umsetzung der partiellen Virtualisierung und die Isolation der Adressräume der einzelnen Prozessinkarnationen muss die Hardware weiterhin geeignete Speicherschutzmechanismen zur Verfügung stellen, zum Beispiel in Form einer *Speicherverwaltungseinheit (MMU)* (engl. *memory management unit*) oder einer *Speicherschutzeinheit (MPU)* (engl. *memory protection unit*).

Eigenschaften der Anwendung

Ein replikdeterministisches Verhalten der Replikate ist eine Grundvoraussetzung für die Anwendbarkeit eines exakten Mehrheitsentscheids: Korrekt arbeitende Replikate liefern unter gleichen Eingaben identische Ergebnisse und weisen insgesamt einen konsistenten beobachtbaren Zustand auf. Die zu replizierende Anwendung besitzt hierfür wohldefinierte Eingänge und Ausgänge – weitergehende Abhängigkeiten mit der Außenwelt sind ausgeschlossen. Die Ablaufstruktur ist weiterhin zyklisch und statisch und folgt dem Muster: Eingaben lesen, Berechnungen durchführen, Ausgaben schreiben. Die Berechnungen erfolgen dabei unabhängig und erlauben einen *unterbrechungsfreien Durchlauf* (engl. *run-to-completion*) der Replikate. Setzt sich die Anwendung aus mehreren Teilaufgaben zusammen, so entsprechen diese wiederum den gleichen Vorgaben.

Dieses *Eingabe-Verarbeitung-Ausgabe*-Prinzip *(EVA)* stellt das vorherrschende Konzept für den Entwurf von Regelungsanwendungen dar [FPW97, KG02, ÅW11]. Gängige Entwurfswerkzeuge wie MATLAB/SIMULINK[2] generieren entsprechend dedizierten Programmcode für das Setzen der Eingänge, die Durchführungen des Regelschritts sowie die Ausgabe des Stellwertes am Ende der Berechnung [Tew02]. Entsprechend eignen sich Regelungsanwendungen im Allgemeinen für die Replikation unter den zuvor genannten Annahmen.

System- und Fehlermodell

Es gelten die für TMR-Systeme bewährten Grundannahmen [VN56]: Replikate weisen dedizierte Schnittstellen und einen unterbrechungsfreien Durchlauf auf. Pro Fehlerbereich (TMR-Stufe) treten jeweils nur *Einzelfehler* auf – geeignete Isolationsmaßnahmen der Laufzeitumgebung verhindern zudem die Fehlerausbreitung. Darüber hinaus *verwirft* CORED *jedoch den angenommenen Perfektionskern* aus Replikator und Mehrheitsentscheider und ordnet diese bislang als fehlerfrei angenommenen Teile eigenen Fehlerbereichen zu.

3.3 CORED: Ansatz und Methodik

Die erweiterte Problemanalyse in Kapitel 3.1 zeigt anschaulich die sich aus der softwarebasierten Replikation auf einer unzuverlässigen Hardwareplattform ergebenden Probleme auf. Auf der einen Seite bietet die replizierte Ausführung zwar die angestrebte vollständige Fehlererfassung und erlaubt darüber hinaus die anwendungstransparente Umsetzung von Fehlertoleranz auf Ebene des Betriebssystems. Auf der anderen Seite erweisen sich die kritischen Fehlerstellen und Lücken in der Replikation jedoch als teilweise unterschätzter Fallstrick für die praktische Umsetzbarkeit softwarebasierter Redundanz. Der Kern dieser Problematik liegt in dem notwendigen Vertrauen in die Verlässlichkeit der übrigen Bausteine der Replikation. Insbesondere die Mehrheitsentscheider, als sprichwörtliche Wächter über die Replikation, stellen einen ernst zu nehmenden Zuverlässigkeitsengpass dar. Ihre Kolokalisierung auf derselben unzuverlässigen Hardware führt entsprechend zu der bereits von Juvenal (Satire 6, 347 f.) aufgeworfenen, in diesem Fall ernsthaften, *Frage nach den Wächtern der Wächter*.

[2] Hersteller Webseite: `www.mathworks.de/products/simulink/`, Abruf: 18.11.2013

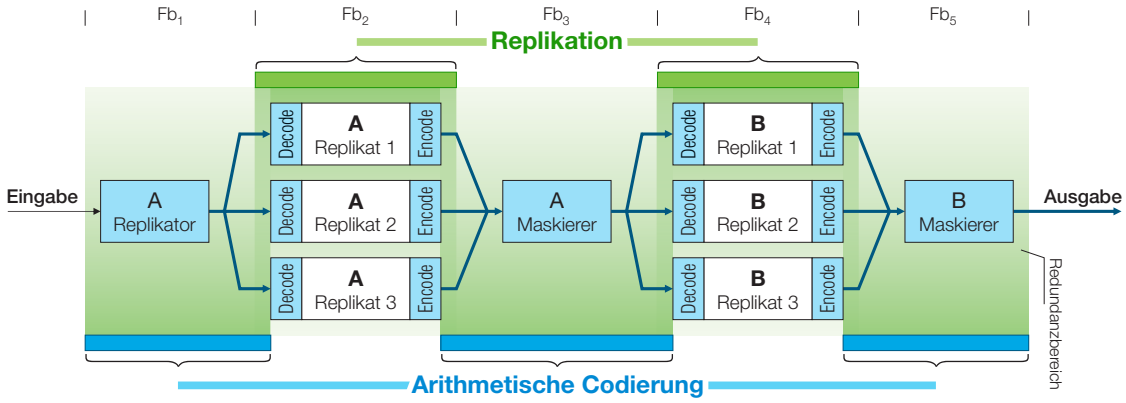


Abbildung 3.3: Übersicht über den CoRed-Ansatz.
Wie der Name bereits impliziert, verbindet der *kombinierte Redundanzansatz* CoRed die beiden Fehlertoleranztechniken *Replikation* und *arithmetische Codierung*. Der Ansatz nimmt die Eingabereplikation sowie die Maskierer aus dem Perfektionskern und schützt diese nicht replizierbaren Teile mittels Codierung. Durch das Zusammenspiel entsteht ein geschlossener Redundanzbereich mit vollständiger Fehlererkennung.

Übersicht und Entwurfskonzept

Der CoRed-Ansatz löst dieses Problem durch die namensgebende Kombination aus *redundanter Ausführung* und *arithmetischer Codierung*. Letztere lässt sich auch in Bereichen einsetzen, die sich nicht weiter replizieren lassen. Abbildung 3.3 gibt einen schematischen Überblick über das zugrunde liegende Entwurfskonzept und das effektive Zusammenspiel der beiden Techniken. Sowohl die Mehrheitsentscheider als auch die Replikatoren werden aus dem angenommenen Perfektionskern genommen und bilden nun eigenständige Fehlerbereiche. Ausgehend von der replizierten Ausführung der Anwendungskomponenten erweitert CoRed den Redundanzbereich schrittweise durch eine angepasste arithmetische Codierung der *Eingangsreplikation* (vgl. Kapitel 3.4) sowie der gesamten Datenflüsse zwischen den Replikaten und zu den Ausgängen. Durch den Entwurf *zuverlässiger Mehrheitsentscheider* (vgl. Kapitel 3.5) lassen sich schließlich die verbleibenden nicht replizierbaren Teile des Systems mit entsprechender Fehlertoleranz versehen. In der Summe ergeben die verschiedenen und sich überlappenden Redundanzbereiche aus Replikation und arithmetischer Codierung einen vollständigen und zusammenhängenden Redundanzbereich von den Eingängen der sicherheitskritischen Anwendung bis zu deren Ausgängen. CoRed eliminiert letztlich die kritischen Fehlerstellen vollständig und bietet damit das technische Rüstzeug für eine gezielte und systematische Beeinflussung der Zuverlässigkeit durch softwarebasierte Redundanz.

Replikation

Für die Replikation nutzt CoRed die durch das Betriebssystem bereitgestellte partielle Virtualisierung der Hardware, um die geforderten räumlichen und zeitlichen Isolationseigenschaften der Prozesse zu erreichen. Im Zuge der Replikation werden die Prozessinkarnationen der Anwendung vervielfältigt, wobei sich deren Art und Umfang nach den in Kapitel 3.2.2 genannten Anforderungen des Systemmodells richtet: Die entstehenden Replikate besitzen insbesondere dedizierte Schnittstellen für die Eingänge beziehungsweise Ausgänge und erlauben darüber hinaus einen unterbrechungsfreien Durchlauf.

CoRed fasst die Replikate darüber hinaus in die notwendige Replikationsinfrastruktur ein und stellt die Codierung und Decodierung an deren Eingängen beziehungsweise Ausgängen sicher.

ANBD Codierung

```
1 function ENCODE(v, A, B, D)
2     return v · A + B + D          ⇝ Codierter Wert v_c
3 end function
```

ANBD Decodierung

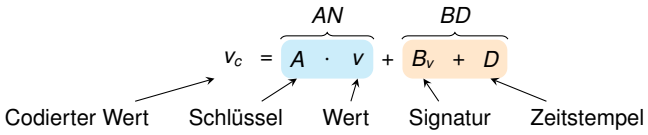
```
4 function DECODE(v_c, A, B, D)
5     if v_c > v_{c,max} then SIGNAL_DUE()
6     if v_c mod A ≠ (B + D) then SIGNAL_DUE()
7     return (v_c − (B + D)) div A          ⇝ Wert v
8 end function
```

Programmausschnitt 3.1: Funktionen für die arithmetische Codierung und Decodierung von Daten.
Von Forin [For89] vorgeschlagene Codierung (links) beziehungsweise Decodierung (rechts) eines Wertes *v*. Die Fehlererkennung erfolgt im Zuge der Decodierung durch die Überprüfung auf einen möglichen Rest in Zeile 6.

Die verbleibenden Elemente der Replikation wie die obligatorischen Mehrheitsentscheider und Replikatoren wandern ihrerseits in entsprechende Isolationsdomänen um eine Fehlerausbreitung zwischen den einzelnen Komponenten zu verhindern.

Arithmetische Codierung

Für die arithmetische Codierung der nicht replizierbaren Bereiche setzt der CoRed-Ansatz auf die von Forin [For89] vorgeschlagenen und in Kapitel 2.2.2 bereits näher betrachteten ANBD-Codes. Von den zur Verfügung stehenden Codierungstechniken eignen sich diese insbesondere aufgrund der konzeptionell vollständigen Fehlererfassung von Operanden-, Operator- und Berechnungsfehlern. Die Codierungsvorschrift setzt sich im Einzelnen zusammen aus:



Die eigentliche Fehlererkennung beruht hierbei auf dem *AN*-Anteil der Codierung. Dieser dient der Erzeugung eines Abstandes um *A* und ermöglicht somit erst die Erkennung von fehlerhaften Codewörtern. Gerade mit Blick auf die komplexen Zusammenhänge zwischen der Zuverlässigkeit der Hardware und der effektiven Fehlerrate der Anwendung aus Kapitel 3.1.2 ist die Effektivität der Codierung von hoher Relevanz für den verfolgten Ansatz. Dieser Aspekt ist Gegenstand einer ausführlichen Untersuchung der Codierungseigenschaften in Kapitel 4. Für den folgenden Entwurf steht hingegen zunächst der *BD*-Anteil im Mittelpunkt. Dieser dient vereinfacht ausgedrückt der gezielten Manipulation und Verfälschung des Codewortes, um neben den reinen Datenfehlern auch die anderen Fehlerklassen in der Decodierungsphase sichtbar zu machen. Um dieses Ziel zu erfüllen, muss CoRed die eingesetzten Signaturen (*B*) und Zeitstempel (*D*) geeignet instrumentalisieren und einsetzen. Die zentralen Herausforderungen liegen hier in der in Kapitel 3.5 folgenden, codierten Darstellung der Mehrheitsentscheider.

Der kombinierte Redundanzansatz (CoRed)

CoRed nutzt eine Kombination aus Replikation von Prozessinkarnationen und ANBD-Codes, um eine durchgehende Fehlererkennung der softwarebasierten Redundanz zu erreichen. Das geeignete Zusammenspiel der Techniken eliminiert die Zuverlässigkeitsengpässe und überwacht die Mehrheitsentscheider als Wächter der Replikation.

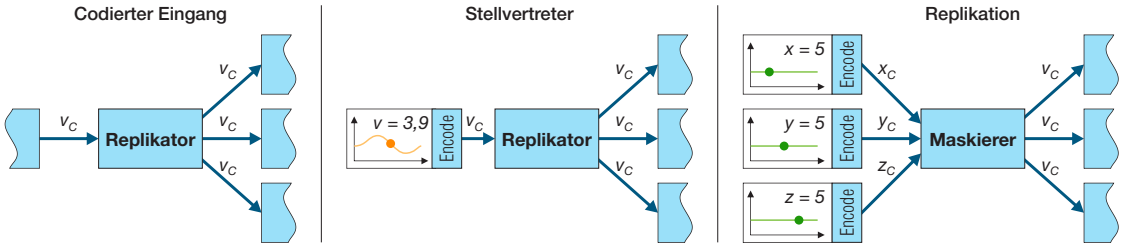


Abbildung 3.4: Verschiedene Arten der Eingangsreplikation in CoRed.
Abhängig von der Art der Quelle stehen unterschiedliche Konzepte für die Eingangsreplikation zur Verfügung. Bereits *codierte Eingänge* koppeln verschiedene CoRed-Blöcke und Anwendungen und erlauben so die einfache Erweiterung des Redundanzbereichs. *Stellvertreter* (engl. *proxy*) übernehmen die Datenakquise bei Eingängen, die nichtdeterministisch oder nicht seiteneffektfrei sind. Ziel ist hier die Codierung der Daten zum frühestmöglichen Zeitpunkt. Deterministische Quellen erlauben ein mehrfaches Abfragen und damit die *Replikation* der Information. Vor der Weiterverwendung werden die redundanten Daten durch einen Mehrheitsentscheider geprüft.

3.4 Eingang-zu-Ausgang Fehlererkennung

Das Ziel des CoRed-Ansatzes ist die Bereitstellung einer durchgehenden Fehlererkennung von den Eingängen des Systems bis zu dessen Ausgängen. Konkret erfordert dies die Anbindung der sicherheitskritischen Anwendung an ihre Außenwelt. Der grundlegende Lösungsansatz ist dabei, neben der Replikation der Verarbeitungselemente, die durchgehende Codierung der nicht replizierbaren Datenströme von ihrem initialen Auftreten bis zum Erreichen der replizierten Ausführung. Die Replikate erhalten ihre Eingaben daher grundsätzlich in einer codierten Form und geben ihre Ausgaben ebenso codiert an die Außenwelt ab. Die beiden in Programmausschnitt 3.1 dargestellten Basisfunktionen für die Codierung (`encode()`) beziehungsweise Decodierung (`decode()`) markieren damit letztlich die beiden äußersten Punkte der in Software realisierbaren Fehlertoleranz und damit auch der durch CoRed geschützten sicherheitskritischen Anwendung.

3.4.1 Eingangsreplikation

Für die replizierte Ausführung ist eine geeignete, den geforderten Replikdeterminismus erhaltende, Vervielfältigung der Eingänge obligatorisch. Im Rahmen der konsekutiven Ausführung der replizierten Bereiche ist diese Eigenschaft konzeptbedingt durch den codierten Mehrheitsentscheider gegeben. Abbildung 3.3 zeigt diesen Fall in der Verbindung der Replikatblöcke A und B. Die Daten stehen hier bereits in ihrer codierten Form zur Verfügung und können direkt im Anschluss an die Mehrheitsbildung sicher auf die nachfolgenden Replikate verteilt werden. Im Allgemeinen lassen sich die in Abbildung 3.4 dargestellten und im Folgenden erläuterten Varianten der Eingangsreplikation unterscheiden.

Codierter Eingang

Ein *codierter Eingang* stellt den zuvor beschriebenen, optimalen Fall der Eingangsreplikation dar, welcher sich letztlich auf die bloße Vervielfältigung des codierten Eingangswertes reduzieren lässt. Im Idealfall entsprechen nicht nur die Verbindungen zwischen den replizierten Domänen diesem Schema, sondern auch alle weiteren Anknüpfungspunkte an die Außenwelt – die Verarbeitungskette der sicherheitskritischen Anwendung beginnt entsprechend bereits mit *codierten* Eingängen. Dieses Szenario lässt sich durch den Einsatz entsprechender Datenquellen realisieren. Im einfachsten Fall handelt es sich hierbei um die Ausgänge einer anderen CoRed-Anwendung. Ebenso ist

auch eine direkte Einbeziehung einer, durch die in Kapitel 2.2.2 genannten Techniken transformierten, ANBD-codierten Anwendung möglich. Weitere Beispiele für derartige Erweiterungen sind die entsprechende Absicherung der lesenden Systemaufrufe durch das Betriebssystem oder eine sensorseitige Fehlererkennung und Codierung der Messwerte als eine externe beziehungsweise systemübergreifende Maßnahme.

Stellvertreter

Im Allgemeinen ist zumindest an den Schnittstellen zur Hardware und der Peripherie nicht mit einer direkten Bereitstellung codierter Eingänge zu rechnen. In diesem Fall ist der Einsatz von *Stellvertretern* (engl. *proxy*) eine gängige Vorgehensweise zur Anbindung regulärer Eingänge an die CoRed-Schutzdomäne. Ein Stellvertreter führt alle für eine korrekte Erfassung der Eingangsdaten notwendigen, nicht replizierbaren Aktionen durch. Ein charakteristisches Beispiel ist das Auslesen von Hardwareregistern: Der Wert eines *Analog-Digital-Wandlers* (engl. *analog-to-digital converter*) wird typischerweise kontinuierlich durch die Hardware aktualisiert und lässt sich daher nicht mehrfach abfragen. Folglich liest der Stellvertreter den Eingang einmalig und codiert die Daten direkt im Zuge des Auslesens, bevor diese an die Replikate verteilt werden. In ähnlicher Weise lassen sich beispielsweise auch komplexere Treiber oder Systemaufrufe an die Replikation ankoppeln. Insgesamt haben Stellvertreter den Vorteil, dass sie universell einsetzbar sind und kein deterministisches Verhalten der eigentlichen Datenquelle erfordern. Demgegenüber ist eine Fehlererkennung erst ab dem Punkt der Codierung gegeben. Insbesondere bei seiteneffektbehafteten Quellen stellt die sofortige Codierung der Daten beim Erreichen der Schutzdomäne jedoch die einzige Art der Ankopplung dar. Die Bereitstellung geeigneter Stellvertreter für die Eingänge des Systems wie auch für die von der Anwendung benötigten lesenden Systemaufrufe ist daher ein integraler Bestandteil der CoRed-Dienstschicht (vgl. Kapitel 3.7.1).

Replikation

In bestimmten Fällen bietet sich auch die *Replikation* der Eingänge in Form temporaler Diversität und Informationsredundanz an. Hierbei wird der Eingang *mehrfach* ausgelesen und vor der Weiterverteilung einem Mehrheitsentscheid unterzogen. Diese Art der Eingangsreplikation schiebt die Grenzen der Fehlererkennung durch die redundante Erfassung der Eingangsdaten bis an die Datenquelle. Letztlich entspricht dieses Vorgehen einer virtuellen Replikation der Quelle. Wie schon für die Replikate, müssen hierfür die selben Anforderungen hinsichtlich des Replikdeterminismus des Eingangs gelten. Dieser muss entsprechend in dem gegebenen Betrachtungszeitraum einen deterministischen Wert aufweisen und sich seiteneffektfrei abfragen lassen. Ein typisches Beispiel sind Zustandsparameter, die sich erst durch ein weiteres Voranschreiten der Ausführung verändern (dürfen).

Nichtdeterministische Datenquellen

Unglücklicherweise erfüllen in der Praxis die wenigsten Datenquellen diese Anforderungen an ein deterministisches Verhalten. Ein typisches Beispiel stellen die für die Regelung eingesetzten Sensoren dar. Diese technischen Bauteile dienen der Erfassung von physikalischen oder chemischen Eigenschaften [Fra10] und weisen eine inhärente Unsicherheit hinsichtlich der Messwertausgabe auf, zum Beispiel in Form von Rauschen. Abbildung 3.5 zeigt dies exemplarisch anhand eines Sensorsystems, bestehend aus drei Gyroskopen. Ein mehrfaches Abtasten eines Signaleingangs führt hier mit hoher Wahrscheinlichkeit zu divergierenden Ergebnissen und einem Verlust des Replikdeterminismus. In der Folge ist der eingesetzte Relativtest auf Übereinstimmung nicht mehr in

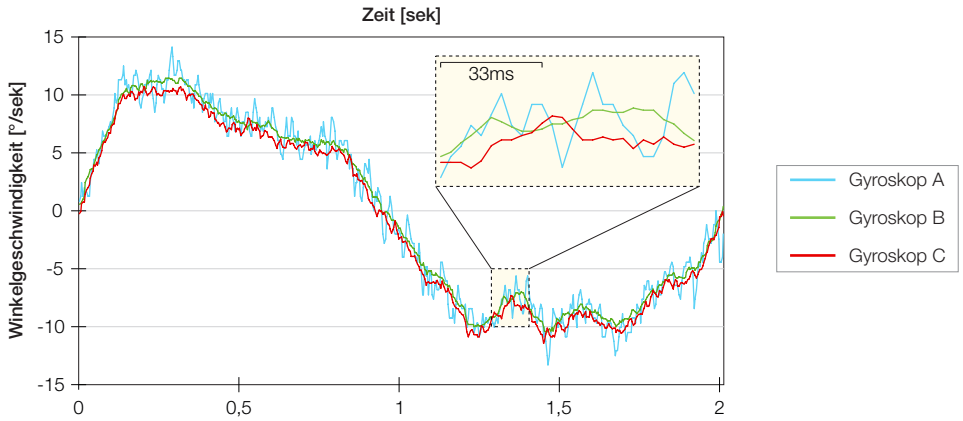


Abbildung 3.5: Typische Abweichung der Eingangssignale eines Sensorsystems.
Ein dreifach redundantes Sensorsystem aus Winkelmessern (Gyroskop). Die Sensoren messen die gleiche physikalische Größe, weichen jedoch durch das immanente Rauschverhalten geringfügig voneinander ab. Auch ein mehrfaches Abtasten des selben Eingangs führt so zu unterschiedlichen Werten. Aus Sicht der Eingangsreplikation stellen Sensoren daher üblicherweise nichtdeterministische Quellen dar.

der Lage die Varianten zuverlässig einer Konsensmenge zuzuordnen. Diese in der Literatur häufig *Problem des konsistenten Vergleichs* (engl. *consistent comparison problem*) [BKL89, Pul01] genannte Fragestellung lässt sich entsprechend mit dem in Abbildung 3.4 gezeigten Replikationsmuster nicht mehr auflösen.

Eine gängige Lösung dieses Problems ist daher die Umstellung des Vergleichs auf *Ähnlichkeit* anstelle der bisher angestrebten Übereinstimmung. Statt eines exakten Mehrheitsentscheiders kommen in der Folge *Akzeptanzmaskierer* (engl. *acceptance voters*) zum Einsatz [LCE89, Pul01]. Der *Intervallentscheider* stellt hierbei eine einfache Realisierung für den näherungsweisen Vergleich dar. Der Mehrheitsentscheid erfolgt hier nicht mehr auf den eigentlichen Varianten (x, y, z), sondern auf dem Abstand δ zwischen diesen:

$$\begin{aligned} d(x,y) &= |y-x| = \delta_1 \\ d(x,z) &= |z-x| = \delta_2 \\ d(y,z) &= |y-z| = \delta_3 \end{aligned} \tag{3.3}$$

Der Maskierer ordnet die Varianten entsprechend der errechneten Abstände δ_i der Konsensmenge zu. Der Schwellenwert für einen als gültig betrachteten Abstand ist hierbei *anwendungsspezifisch*. Bezogen auf das Beispiel in Abbildung 3.5, ergibt sich der zulässige Abstand zwischen den Messerwerten beispielsweise aus dem Rauschen des Sensors sowie der maximalen Änderung der Winkelgeschwindigkeit innerhalb des Abtastintervalls. Neben der Intervallentscheidung existieren je nach Anwendungsszenario noch weitere Umsetzungsalternativen wie beispielsweise die Median- oder Kugelentscheidung [Ech90]. Wie der Name bereits andeutet, lassen sich die Akzeptanzmaskierer im Allgemeinen auf eine implizite Kombination aus Akzeptanz- und Relativtest zurückführen. Die Entscheidung, ob ein zuvor festgelegter Abstandswert zwischen zwei Varianten eingehalten wird, entspricht einem Akzeptanztest dieser Distanzberechnung. Die Mehrheitsfindung erfolgt im Anschluss wiederum durch einen einfachen Relativtest, welcher die Antworten der Akzeptanztests vergleicht.

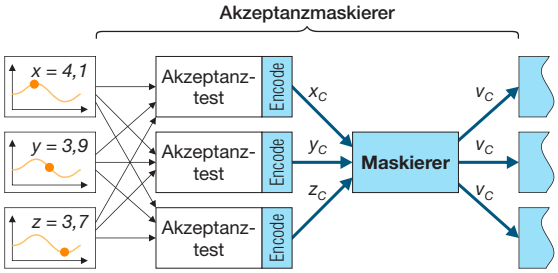


Abbildung 3.6: Akzeptanzmaskierer für die Eingangsreplikation nichtdeterministischer Quellen in CoRed.
Selbst leicht voneinander abweichende Eingaben führen zu einem Verlust des Replikdeterminismus. *Akzeptanzmaskierer* erlauben komplexere Vergleichsoperatoren, beispielsweise auf Gültigkeitsintervalle. CoRed unterteilt diese in den anwendungsabhängigen und replizierbaren Akzeptanztest und den regulären, auf exakter Übereinstimmung basierenden Maskierer.

Akzeptanzmaskierer in CoRed

Den Gestaltungsgrundsätzen aus Kapitel 3.3 folgend, teilt CoRed die Akzeptanzmaskierer entsprechend in einen anwendungsspezifischen Akzeptanztest sowie einen nachgeschalteten regulären Mehrheitsentscheider auf. Abbildung 3.6 zeigt das entsprechende Entwurfsschema. Die Datenquelle ist in diesem Fall redundant vorhanden beziehungsweise wird mehrfach ausgelesen. Die Ergebnisse werden anschließend an die Akzeptanztests weitergeleitet. Da diese in jedem Fall hochgradig anwendungsabhängig sind, erfolgt die Ausführung der uncodierten Akzeptanztests wiederum repliziert, um eine effektive Fehlererkennung zu gewährleisten. Nach diesem Punkt gleicht die Eingangsreplikation für nichtdeterministische Eingänge wiederum der regulären Replikation aus Abbildung 3.4.

Insgesamt ist bei sicherheitskritischen Anwendungen davon auszugehen, dass die notwendige Sensorik bereits physikalisch redundant ausgelegt ist, da diese technischen Bauteile häufig im besonderen Umfang Umwelteinflüssen ausgesetzt sind und im Vergleich mit dem eigentlichen Rechensystem in der Regel eine deutlich niedrigere Zuverlässigkeit aufweisen [Fra10]. In diesen Fällen stehen die notwendigen Akzeptanztests für den Umgang mit den redundanten Eingängen bereits zur Verfügung und müssen lediglich in geeigneter Weise mit den CoRed-Eingangskomponenten verschaltet werden.

Grundsätzlich lassen sich Akzeptanzentscheider anstatt an den Eingängen auch an den Ausgängen der Replikation einsetzen. In diesem Fall dienen sie einer unscharfen Bewertung der Berechnungsergebnisse. Da hierbei der interne Zustand der Replikate jedoch voneinander abweichen kann, ist der Replikdeterminismus nicht mehr gewährleistet. Von einem entsprechenden Entwurf wird daher üblicherweise abgesehen. Ein durchaus valides Einsatzfeld der Akzeptanzmaskierer sind dagegen große und komplexe Replikatausgänge beziehungsweise Datenströme. Um die für den Einzelvergleich der Werte benötigte Bandbreite zu mindern, lässt sich der rechenaufwendige Teil des Zustandsvergleichs (engl. *state comparison*) wiederum in einen Akzeptanztest auslagern. Die kritische Mehrheitsentscheidung und die Auswahl des Gewinners basieren wiederum auf dem geschützten CoRed-Mehrheitsentscheider.

3.4.2 Ausgangsvergleich und Konsolidierung

Vor der Ausgabe eines Ergebnisses an die Außenwelt erfolgt in jedem Fall die Konsolidierung der replizierten Ausführung durch den Mehrheitsentscheider. Dieser tritt, abhängig davon, ob es sich um ein Zwischen- oder Endergebnis handelt, als Synchronisationspunkt und *Koppler* zwischen Replikationsdomänen oder als *Endpunkt* auf (vgl. Maskierer A und B in Abbildung 3.3).

Erweiterte Grenzen der Replikation

Bereits der Einsatz von codierten Datenflüssen schließt die Lücken in der Fehlererfassung *innerhalb* der Replikation – eine Mutation von (Zwischen-)Ergebnissen ist nicht mehr möglich. Die verschiedenen Arten der Eingangsreplikation erlauben zudem die Ausweitung des Redundanzbereiches bis an die in Software realisierbaren Grenzen. Der Einsatz codierter Eingänge und Ausgänge ermöglicht letztlich die Überwindung der lokalen Grenzen der Replikation und die Ausweitung des Redundanzbereiches sogar *über Systemgrenzen hinweg*.

Stellvertreter

Wie bereits erwähnt, repräsentiert die Decodierung den finalen Endpunkt einer durch CoRed geschützten Ausführungskette. Entsprechend folgt auf den Mehrheitsentscheider ein letztes `decode()` bevor das Ergebnis den Redundanzbereich der arithmetischen Codierung verlässt. Wie schon bei der Eingangsreplikation, übernimmt der Ausgabeblock wiederum die Funktion eines Stellvertreters. Dieser führt letztlich den für die tatsächliche Ausgabe notwendigen Epilog aus, zum Beispiel durch Setzen des entsprechenden Ausgaberegisters der Hardware oder die Übergabe an einen schreibenden Systemaufruf. Neben der fallspezifischen Implementierung des Stellvertreters, dient die CoRed-Dienstschicht (vgl. Kapitel 3.7.1) wiederum der Emulation der gängigen Systemschnittstellen und Treiber.

Codierter Ausgang

Eine direkte Ausgabe der Ergebnisse am Ende der replizierten Ausführung ist in der Regel nicht zweckmäßig, da hiermit sowohl die Konsolidierung wie auch die Fehlererkennung durch den Mehrheitsentscheider entfallen würde. Um dennoch eine Fortführung der Fehlererkennung zu ermöglichen, stellt CoRed analog zu den codierten Eingängen auch *codierte Ausgänge* zur Verfügung. Über die Verknüpfung von CoRed-Anwendungen hinaus, lässt sich hiermit der Redundanzbereich über die Grenzen der eigenen Anwendung und sogar des Rechensystems erweitern. So kann beispielsweise der Schutz vor transienten Fehlern bis zu den Aktoren erhalten bleiben, indem die Stellwerte der Regelung codiert an diese übertragen werden. Ein Aktor kann somit die Korrektheit des Ergebnisses sowie der Mehrheitsentscheidung an sich eigenständig prüfen und den Wert erst kurz vor seiner tatsächlichen Verwendung decodieren. Dieses Vorgehen lässt sich neben der Peripherie auch auf verteilte Echtzeitsysteme im Allgemeinen anwenden.

3.5 Zuverlässige Mehrheitsentscheider

Die technische Umsetzung eines Mehrheitsentscheiders erweist sich prinzipiell als vergleichsweise einfach. Programmausschnitt 3.2 zeigt dies am Beispiel der Funktion `vote()` für die drei *Varianten* (engl. *variants*) x, y und z. Die Bestimmung einer qualifizierenden Mehrheit erfolgt hier durch eine Folge von *bedingten Verzweigungen* (engl. *conditional branch*) (Zeilen 2, 3, 8 und 10), welche zur Auswahl einer Variante und damit zu deren Rückgabe als Gewinner (*win*) führt. Ist eine Mehrheitsfindung nicht möglich, erfolgt die Signalisierung eines erkannten, nicht behebbaren Fehlers. Grundsätzlich weist der Aufbau des Mehrheitsentscheiders dabei bereits ein gewisses Maß an inhärenter Redundanz auf. Durch den Vergleich aller Varianten untereinander lässt sich beispielsweise eine Abweichung von y sowohl durch den Vergleich in Zeile 2 wie auch durch den Vergleich in Zeile 8 entdecken.

```
1  function VOTE(x, y, z)
2      if x = y then
3          if x = z then
4              return win ← x                ⇝ Alle Ergebnisse identisch
5          else
6              return win ← x                ⇝ z weicht ab
7          end if
8      else if y = z then
9          return win ← y                    ⇝ x weicht ab
10     else if x = z then
11         return win ← x                    ⇝ y weicht ab
12     else
13         SIGNAL_DUE()                      ⇝ Keine Entscheidung möglich
14     end if
15 end function
```

Programmausschnitt 3.2: Einfacher Mehrheitsentscheider.
Funktion `vote()` zur Bestimmung der Mehrheit der drei Varianten *x*, *y* und *z*. Die Mehrheitsfindung erfolgt durch eine einfache Folge von Vergleichen und bedingten Verzweigungen. Im Falle einer qualifizierenden Mehrheit (2 aus 3) wird eines der mehrheitsbildenden Ergebnisse ausgewählt und als Gewinner (*win*) zurückgegeben. Ist keine Entscheidung möglich, wird ein erkannter, nicht behebbarer Fehler (DUE) signalisiert.

Trotz des einfachen Aufbaus und der inhärenten Redundanz sind die Auswirkungen transienter Fehler auf den Mehrheitsentscheider signifikant. Im Vorgriff auf die experimentelle Evaluation in Kapitel 4 erweisen sich über *20 Prozent* aller effektiven Fehler als schwerwiegend und führen zu einem Fehlerverhalten des ungeschützten Mehrheitsentscheiders sowie unerkannten Datenfehlern. Das grundlegende Konzept des CoRed-Ansatzes ist daher die Bereitstellung eines *zuverlässigen Mehrheitsentscheiders* (engl. *dependable voter*), welcher in der Lage ist die Mehrheitsbildung auf codierten Werten durchzuführen, ohne den Schutz der Codierung zu verlieren.

3.5.1 Arithmetische Codierung – Vergleichsoperatoren

Für die Umsetzung eines solchen zuverlässigen Mehrheitsentscheiders ist zunächst die arithmetische Codierung der Funktion `vote()` aus Programmausschnitt 3.2 und insbesondere der enthaltenen Vergleichsoperatoren und bedingten Verzweigungen notwendig.

Kontrollflussabhängige Signaturanpassung

Forin [For89] schlägt hierfür die in Programmausschnitt 3.3 dargestellte codierte Variante einer bedingten Verzweigung vor. In diesem Beispiel wird der uncodierte Programmausdruck (links) in eine durch ANBD-Codes geschützte Version (rechts) überführt. Die grundlegende Idee ist hierbei die schrittweise Herleitung einer vom Ausführungspfad unabhängige Signatur am Ende des Vergleichs. Diese ist jedoch nur bei einer korrekten Abfolge der Instruktionen erreichbar, da sich die einzelnen Berechnungsschritte zwischen den beiden Zweigen unterscheiden.

Die Codierung der bedingten Verzweigung erfordert hierfür zunächst die Bereitstellung der *Sprungsignaturen* B_{true} und B_{false} für die beiden möglichen Ausführungspfade – Forin nutzt hierfür im Wesentlichen die Signaturen der beteiligten Operanden (hier x_c und y_c). In einem ersten, uncodierten Berechnungsschritt wird in Zeile 1 das Ergebnis der Bedingung ($x \geq y$) ermittelt und die Korrektursignatur B_{cond} entsprechend gesetzt. Das Ziel der Codierung ist nunmehr die Erzeugung einer konstanten und von der bedingten Ausführung unabhängigen Zielsignatur am Ende der Verzweigung (Zeile 11). In diesem Beispiel wird die Signatur von y_c bei der Durchführung des

Original

```
   State: x, y

1  if (x ≥ y) then
2      y ← x − y
3  else
4      y ← x + y
5  end if
```

ANBD-codiert

```
   Requires: B_x, B_y, B_true, B_false ⇝ Konstante Signaturen für Operanden und Zweige
   State: x_c, y_c, B_cond

1  if (DECODE(x_c) ≥ DECODE(y_c)) then B_cond ← B_true else B_cond ← B_false
2
3  if (DECODE(x_c) ≥ DECODE(y_c)) then
4      y_c ← x_c − y_c                                  ⇝ Signatur: B_x − B_y
5  else
6      y_c ← x_c + y_c                                  ⇝ Signatur: B_x + B_y
7      y_c ← y_c − (B_x + B_y) + (B_x − B_y)            ⇝ Signaturanpassung: B_x − B_y
8      y_c ← y_c − B_false + B_true                     ⇝ Verzweigung signieren
9  end if
10
11 y_c ← y_c + B_cond                  ⇝ Signaturanpassung, Sollwert: B_x − B_y + B_true
```

Programmausschnitt 3.3: Arithmetische Codierung einer Größer-Gleich-Verzweigung.
Von Forin [For89] vorgeschlagene Codierung eines Größer-Gleich-Vergleichs (links die uncodierte Variante). Ausgangspunkt ist die Auswahl einer statischen Signatur (B_{true}, B_{false}) für den erwarteten Verlauf der Verzweigung (Zeile 1). Diese Auswahl wird am Ende der Verzweigung (Zeile 11) genutzt um eine von dem Verlauf unabhängige, feste Signatur zu erhalten. Diese entspricht in diesem Beispiel der codierten Subtraktion aus Zeile 4 sowie der statischen Signatur B_{true}. Um dieses Soll auch im `else`-Fall zu erreichen, werden die Signaturen hier entsprechend angepasst (Zeile 7f). *Beispiele aus bzw. nach: [For89, Sch11b]*

`if`-Zweigs als Sollwert angenommen. Dieser setzt sich zusammen aus der Signatur der Subtraktion ($B_x - B_y$, Zeile 4) sowie der Sprungsignatur (B_{true}). Erst im zweiten Schritt wird die bedingte Verzweigung tatsächlich ausgeführt (Zeile 3–9). Trifft die Bedingung zu, wird die Subtraktion in Zeile 4 durchgeführt und die Signatur von y_c entsprechend gesetzt. Darüber hinaus evaluiert B_{cond} zu B_{true}, womit der Sollwert durch die Signaturanpassung in Zeile 11 wie geplant erreicht wird. Um dieses Ergebnis auch bei Eintreten des `else`-Zweigs herbeizuführen, werden die Signaturen nach der Addition in den Zeilen 7 und 8 geeignet angepasst. Durch die vom Ausführungspfad unabhängige Zielsignatur des Wertes y_c kann dieser anschließend direkt in weiteren codierten Berechnungsschritten genutzt werden. Wird einer der Operanden, eine Berechnung oder die Ausführungsreihenfolge durch einen transienten Fehler korrumpiert, so weicht die Zielsignatur von dem erwarteten Wert ab. In der Folge stellt der codierte Werte y_c mit hoher Wahrscheinlichkeit kein gültiges Codewort mehr dar.

Einschränkungen und Entwurfsalternativen

Das beschriebene Entwurfsmuster für die Codierung der Verzweigungen ist jedoch mit einigen Einschränkungen und Voraussetzungen behaftet. Die bislang bekannten Vergleichsoperatoren für ANBD-Codes beschränken sich auf die von Forin präsentierten codierten Größer-Gleich- ($\geq_c$) beziehungsweise Kleiner-Gleich-Bedingungen ($\leq_c$). Entsprechend lässt sich der codierte Vergleich auf Übereinstimmung ($x_c =_c y_c$) lediglich durch die umständliche Kombination der beiden bekannten Vergleichsoperatoren ($\leq_c$ und $\geq_c$) sowie der codierten Konjunktion ($\wedge_c$) der Einzelergebnisse erreichen [For89, Sch11b]. Darüber hinaus erfolgt die Codierung der Verzweigung bei Forin implizit durch Anpassung der Signaturen der beteiligten Operanden und Berechnungen. Die Überprüfung des Kontrollflusses erfolgt letztlich zusammen mit der generellen Fehlererkennung bei der Decodierung des Berechnungsergebnisses. Im Beispiel in Programmausschnitt 3.3 werden hierfür y_c sowie die Subtraktion in Zeile 4 beziehungsweise die Addition in Zeile 6 geeignet instrumentalisiert. Stehen in den Verzweigungen keine hierfür nutzbaren gemeinsamen Operatoren und Berechnungen zur Verfügung, lassen sich die für die Fehlererkennung notwendi-

gen Signaturberechnungen nicht anwenden. Schiffel [Sch11b] umgeht dieses Problem durch die Einführung einer *Pseudovariablen* (engl. *dummy*), auf deren Basis die Signaturberechnung (z. B. der Sprungsignatur B_{cond}) für die Codierung der Vergleichsoperation erfolgt. Eine Rückführung auf die eigentlichen Berechnungen und deren Ergebnisse beziehungsweise eine Anpassung der Signaturen ist dabei nicht mehr obligatorisch. Von diesem Standpunkt aus erfolgt sowohl die Codierung wie auch die Überwachung von Daten- und Kontrollfluss getrennt voneinander. Schiffel et al. [SSSF10] verallgemeinern diesen Ansatz der codierten Kontrollflussüberwachung in CEP (vgl. Kapitel 2.2.2) mittels dedizierter Signaturen für jeden *Grundblock* (engl. *basic block*) des Kontrollflussgraphen. Hierdurch lassen sich beliebige sequentielle Instruktionsfolgen sowie Verzweigungen codieren. Um Kontrollflussfehler aufzudecken und eine ungewollte Fehlerausbreitung sicher zu vermeiden, vergleicht CEP die Kontrollflusssignaturen zu Beginn jedes Grundblocks mit einer vorausberechneten Sequenzliste. Konzeptbedingt erfordert die Verwaltung der Sequenzen sowie deren Abgleich mit der tatsächlichen Ausführungsreihenfolge eine vertrauenswürdige Prüfinstanz. Für deren Umsetzung sieht CEP spezifische Hardwareerweiterungen wie beispielsweise einen *field-programmable gate array (FPGA)* vor.

Zusammenfassend erweist sich die Codierung der für den Mehrheitsentscheider wichtigen Vergleichsoperatoren und bedingten Verzweigungen als nicht trivial. Zum einen muss der Vergleich auf Übereinstimmung im Allgemeinen auf insgesamt drei codierte Operationen abgebildet werden. Die hierfür zur Verfügung stehenden Ansätze lassen sich entweder aufgrund der für die implizite Kontrollflusscodierung fehlenden Berechnungsschritte nicht direkt anwenden oder benötigen durch die abgetrennte Kontrollflussüberwachung eine als fehlerfrei angenommene Prüfinstanz. Die in Kapitel 2.2.2 untersuchten Techniken der ANBD-Codierung weisen darüber hinaus eine verbleibende Restfehlerrate für unerkannte Datenfehler auf.

3.5.2 Der CoRed-Mehrheitsentscheider

Für die spezifische Aufgabe der Mehrheitsbestimmung erscheinen die zuvor beschriebenen, gängigen Vorgehensweisen daher insgesamt als ungeeignet. Der Entwurf des CoRed-*Mehrheitsentscheiders* erfolgt stattdessen unter Ausnutzung von dessen spezifischen Eigenschaften sowie der Tatsache, dass jede Entscheidung eineindeutig einem bestimmten Kontrollfluss zugeordnet werden kann. Weder die Herbeiführung einer festen Zielsignatur noch die Abspaltung der Kontrollflussüberwachung sind daher das Ziel, sondern vielmehr die Bereitstellung eines Mehrheitsentscheiders, welcher sich von außen betrachtet wie eine *codierte Operation* verhält.

Vereinfachter Vergleich auf Übereinstimmung

Der für die Umsetzung wichtige Vergleich auf Übereinstimmung lässt sich grundsätzlich massiv vereinfachen. Die beiden Werte x und y sind genau dann gleich, wenn die Differenz der codierte Wert x_c und y_c der Differenz der statischen Signaturen B_x und B_y entspricht:

$$\begin{aligned} x = y \overset{gdw.}{\longleftrightarrow} x_c - y_c &= \overbrace{(Ax + B_x + D)}^{x_c} - \overbrace{(Ay + B_y + D)}^{y_c} \\ &= \overbrace{Ax - Ay}^{0 \Leftrightarrow x=y} + B_x - B_y \\ &= B_x - B_y \end{aligned} \tag{3.4}$$

Basierend auf dieser Vereinfachung lässt sich der Mehrheitsentscheid mit einer vergleichsweise geringen Zahl an codierten Operationen realisieren.

```
   Require: B_x, B_y, B_z                                  ⇝ Konstante Signaturen der Operanden

 1 function CoRed_vote(x_c, y_c, z_c)
 2     zero_local_storage()
 3     if (x_c − y_c) = (B_x − B_y) then
 4         if (x_c − z_c) = (B_x − B_z) then
 5             win_c ← apply(x_c, (x_c − y_c) + (x_c − z_c))
 6             return win_c, B_E ←const (B_x − B_y) + (B_x − B_z)      ⇝ Alle Ergebnisse identisch
 7         else
 8             win_c ← apply(x_c, (x_c − y_c))
 9             return win_c, B_E ←const (B_x − B_y)                    ⇝ z_c weicht ab
10         end if
11     else if (y_c − z_c) = (B_y − B_z) then
12         win_c ← apply(y_c, (y_c − z_c))
13         return win_c, B_E ←const (B_y − B_z)                        ⇝ x_c weicht ab
14     else if (x_c − z_c) = (B_x − B_z) then
15         win_c ← apply(x_c, (x_c − z_c))
16         return win_c, B_E ←const (B_x − B_z)                        ⇝ y_c weicht ab
17     else
18         win_c ← ∅
19         signal_due()                                            ⇝ Keine Entscheidung möglich
20     end if
21 end function

22 function apply(v_c, B_dyn)
23     if B_dyn > B_max then signal_due()
24     return v_c + B_dyn
25 end function
```

Programmausschnitt 3.4: Zuverlässiger CoRed-Mehrheitsentscheider.
Der CoRed-Mehrheitsentscheider gleicht zunächst einem regulären Entscheider, arbeitet von außen betrachtet jedoch wie eine codierte Operation ($\textcircled{v}_c$). Er akzeptiert die codierten Varianten x_c, y_c und z_c als Eingabe und liefert den ebenfalls *codierten Gewinner* win_c zusammen mit der *konstanten Sprungsignatur* B_E zurück. Die Überwachung des Kontrollflusses erfolgt durch die Berechnung einer *dynamischen Sprungsignatur* zur Laufzeit sowie deren Anwendung (`apply()`) auf den Gewinner vor dessen Rückgabe (Zeilen 5, 8, 12 und 15). Sowohl die Integrität des Gewinners wie auch der Entscheidung selbst kann anschließend durch einen Vergleich der beiden Signaturen überprüft werden.

Entwurf des CoRed-Mehrheitsentscheiders

Programmausschnitt 3.4 zeigt den resultierenden Entwurf des CoRed-Mehrheitsentscheiders. Die Funktion `CoRed_vote()` entspricht in ihrem Aufbau weitgehend der ungeschützten Variante aus Programmausschnitt 3.2 und akzeptiert die nunmehr codierten Varianten x_c, y_c und z_c. Wiederum erfolgt die Bestimmung einer qualifizierenden Mehrheit sowie die Rückgabe der ausgewählten Variante in Form des ebenfalls codierten Gewinners win_c durch eine Folge von bedingten Verzweigungen (Zeilen 3, 4, 11 und 14). Da die Operanden für den Vergleich nicht decodiert werden müssen, ist deren Datenintegrität grundsätzlich sichergestellt.

Ungeachtet der vollständigen Codierung der Datenflüsse ist eine Korrumpierung des Kontrollflusses und damit ein Versagen der eigentlichen Mehrheitsfindung durch transiente Fehler weiterhin möglich, beispielsweise durch fehlerhafte Sprungentscheidungen in den Vergleichen. CoRed löst dieses Problem durch die Einführung von Sprungsignaturen für die fünf möglichen Endpunkte des Mehrheitsentscheids. Diese sind über die jeweilige *Konsensmenge* (engl. *equality set*) E eindeutig definiert, da zu jeder möglichen Mehrheitsentscheidung nur ein gültiger Pfad existiert. Die Rückgabe in Zeile 16 kann beispielsweise nur erreicht werden, wenn der Wert von y_c von den anderen Varianten abweicht. Aus dieser Betrachtung heraus ergeben sich die folgenden *konstanten*

Signaturen B_E für die Blätter des Entscheidungsbaums:

$$\begin{array}{rcl} \overbrace{\{x_c, y_c, z_c\}}^{\text{Konsensmenge:}E} & \Leftrightarrow & \overbrace{(B_x - B_y) + (B_x - B_z)}^{\text{Konstante Signatur: } B_E} \\ \{x_c, y_c\} & \Leftrightarrow & (B_x - B_y) \\ \{x_c, z_c\} & \Leftrightarrow & (B_x - B_z) \\ \{y_c, z_c\} & \Leftrightarrow & (B_y - B_z) \\ \{\} & \Leftrightarrow & \text{keine Entscheidung} \end{array} \tag{3.5}$$

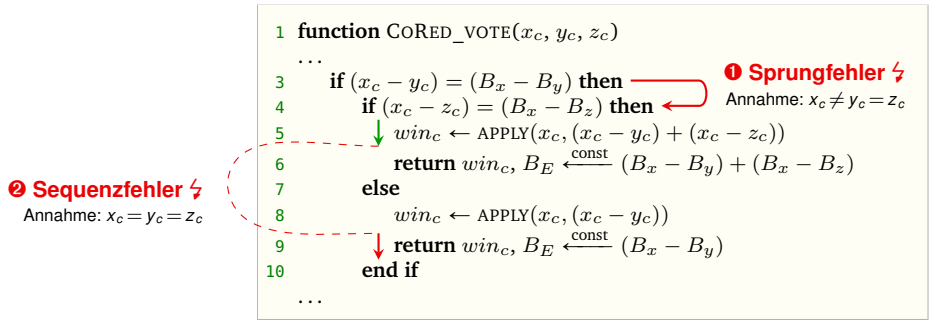
Der Wert von B_E leitet sich hierbei aus den Vergleichsbedingungen für die Übereinstimmung der jeweiligen Varianten der Konsensmenge ab. Stimmen zum Beispiel lediglich x_c und y_c überein, so ergibt sich die Signatur $B_E = B_x - B_y$ entsprechend der hierfür hinreichenden Vergleichsbedingung aus Gleichung 3.4. Die konstanten Signaturen B_E lassen sich zur Übersetzungszeit vorausberechnen und dienen nachfolgenden Komponenten als Indikator für den Ausgang des Entscheids sowie die entstandenen Mehrheitsverhältnisse.

Die konstante Signatur B_E erlaubt die Bestimmung der Identität des Gewinners, jedoch nicht die Überprüfung der Korrektheit der Mehrheitsentscheidung an sich. Aus diesem Grund berechnet der CoRed-Mehrheitsentscheider eine weitere *dynamische Signatur* (B_{dyn}) zur Laufzeit. Diese wird nach demselben Muster wie B_E gebildet und basiert auf den Sprungentscheidungen, die zu einem bestimmten Urteil geführt haben. Stimmen, wie im vorangegangenen Beispiel, alleine x_c und y_c überein, so errechnet sich die dynamische Signatur durch $B_{dyn} = x_c - y_c$ (Programmausschnitt 3.4, Zeile 8). Letztere wird durch die Funktion `apply()` auf die ausgewählte Variante angewendet (Zeilen 5, 8, 12 und 15), bevor diese als der Gewinner win_c zusammen mit der statischen Signatur B_E zurückgeliefert wird (Zeilen 6, 9, 13 und 16). Die Mehrheitsentscheidung war fehlerfrei, wenn die statische und die dynamische Signatur identisch sind. Vereinfacht dargestellt wird der Mehrheitsentscheid wie üblich mittels bedingter Sprünge hergestellt. Vor der Rückgabe des Gewinners wird der an diesem Punkt erwartete Kontrollfluss in Form der hierfür notwendigen dynamischen Signaturberechnungen nachvollzogen und auf den Gewinner codiert. Von außen betrachtet, verhält sich der CoRed-Mehrheitsentscheider wie eine komplexe codierte Operation ($\textcircled{v}_c$):

$$\texttt{CoRed\_vote}(x_c, y_c, z_c) = \underbrace{\begin{cases} x + B_x + D + \overbrace{(B_x - B_y) + (B_x - B_z)}^{\text{Ergebnis der dynamischen Signaturberechnung}} & wenn \quad \overbrace{\{x_c, y_c, z_c\}}^{\text{Konsensmenge (E)}} \\ x + B_x + D + (B_x - B_y) & wenn \quad \{x_c, y_c\} \\ x + B_x + D + (B_x - B_z) & wenn \quad \{x_c, z_c\} \\ y + B_y + D + (B_y - B_z) & wenn \quad \{y_c, z_c\} \\ \texttt{signal\_due()} & wenn \quad \{\} \end{cases}}_{\text{Rückgabewert: } win_c} \tag{3.6}$$

Das Ergebnis des Mehrheitsentscheids hängt naturgemäß von der Konsensmenge der Eingabevarianten (x_c, y_c, z_c) ab. Der Gewinner win_c setzt sich dabei aus der ersten gültigen Variante der Konsensmenge (x_c oder y_c) sowie der dynamischen Signatur B_{dyn} zusammen.

Dieses Verhalten steht zunächst im Gegensatz zu den bekannten codierten Operationen wie beispielsweise der Addition, bei welcher die resultierende Signatur unabhängig von den Eingabewerten vorab bekannt ist. Durch die zusätzliche Rückgabe der statischen Signatur B_E besteht


1 function CoRed_vote(x_c, y_c, z_c)
...
3 if $(x_c - y_c) = (B_x - B_y)$ then
4 if $(x_c - z_c) = (B_x - B_z)$ then
5 $win_c \leftarrow$ apply$(x_c, (x_c - y_c) + (x_c - z_c))$
6 return win_c, $B_E \xleftarrow{\text{const}} (B_x - B_y) + (B_x - B_z)$
7 else
8 $win_c \leftarrow$ apply$(x_c, (x_c - y_c))$
9 return win_c, $B_E \xleftarrow{\text{const}} (B_x - B_y)$
10 end if
...
❶ Sprungfehler
Annahme: $x_c \neq y_c = z_c$
❷ Sequenzfehler
Annahme: $x_c = y_c = z_c$

Programmausschnitt 3.5: Erkennung von Kontrollflussanomalien durch den CoRed-Mehrheitsentscheider.
Grundsätzlich lassen sind zwei Fehlerszenarien unterscheiden: Die korrekte Ausführung des falschen Entscheidungszweiges ist ein Beispiel für einen *Sprungfehler* (①) in einer bedingten Anweisung, beispielsweise ausgelöst durch einen transienten Fehler in den für den Vergleich genutzten Registern. Ein *Sequenzfehler* (②) ist hingegen symptomatisch für eine Verfälschung des Programmzählers und führt zu einer fehlerhaften Ausführung von potenziell mehreren Entscheidungszweigen.

jedoch Kenntnis von der Identität des Gewinners, wodurch eine Decodierung des Ergebnisses trotz der von den Eingaben abhängigen Zusammensetzung ermöglicht wird. Die hierfür nötige inverse Operation `voter_decode()` (nicht dargestellt) zieht entsprechend die konstante Signatur B_E vom Gewinner win_c ab und decodiert die enthaltene Variante x_c oder y_c. Dieses Vorgehen ist für die nachfolgenden Replikate zweckmäßig, da diese direkt auf den uncodierten Nutzdaten weiterarbeiten. Darüber hinaus ist durch die direkte Decodierung im Anschluss an den Mehrheitsentscheid eine eindeutige Diagnose des Mehrheitsentscheiders hinsichtlich eines korrekten Kontroll- *und* Datenflusses gegeben. Ein Fehlverhalten wird entsprechend direkt aufgedeckt und kann durch geeignete Gegenmaßnahmen wie beispielsweise einer Wiederholung des Mehrheitsentscheids adressiert werden.

Soll der Wert von win_c dagegen direkt in codierten Berechnungen weiterverwendet werden, so lässt sich dessen Signatur mittels einer einfachen Recodierungsfunktion auf eine vorherbestimmte Zielsignatur anpassen, ohne die eigentliche Fehlererkennung zu verlieren – dieses Vorgehen imitiert entsprechend das von Forin in Programmausschnitt 3.3 angestrebte Verhalten. Erfolgt in diesem Zusammenhang jedoch keine Überprüfung (Decodierung) des Wertes, ist eine spätere Fehlerdiagnose des Mehrheitsentscheiders ausgeschlossen. Unabhängig davon ist eine Fehlererkennung jederzeit möglich.

Schlüsselelement: Der zuverlässige CoRed-Mehrheitsentscheider

Der CoRed-Mehrheitsentscheider erlaubt die sichere Erkennung von Fehlern in den Ergebnissen wie auch der Mehrheitsfindung an sich. Der Mehrheitsentscheider verhält sich dabei von außen betrachtet wie eine codierte Operation Ⓥ_c. Die Korrektheit einer Entscheidung lässt sich folglich durch eine einfache Decodierung überprüfen.

Fehlererkennung

Die durch den CoRed-Mehrheitsentscheider erreichbare Fehlererkennung gegenüber dem Fehlermodell aus Kapitel 2.1.1 ist konzeptionell vollständig und – im Vorgriff auf die experimentelle Evaluation in Kapitel 4 – technisch zuverlässig: Zunächst lassen sich *Berechnungsfehler* aufgrund der Abwesenheit von codierten Rechenoperationen konzeptbedingt ausschließen. Fehler in den

Operanden (x_c, y_c, z_c) werden durch die allgemeinen Eigenschaften der ANBD-Codierung erkannt. Hierfür sind insbesondere die individuellen Variablensignaturen (B_x, B_y, B_z) sowie der rundenbasierte Zeitstempel D relevant. Es verbleibt die Klasse der *Operatorfehler* und entsprechend ein von außen nicht erkennbares Versagen des CoRed-Mehrheitsentscheiders. Die eingesetzten Sprungsignaturen B_E und B_{dyn} erlauben auch hier eine zuverlässige Fehlererkennung dieser verbleibenden Fehlerklasse und insbesondere auch der möglichen Kontrollflussfehler des Mehrheitsentscheiders.

Die Wirkungsweise der Fehlererkennung im Hinblick auf Kontrollflussanomalien illustriert Programmausschnitt 3.5 am Beispiel zweier Fehlerszenarien: Die korrekte Ausführung des falschen Entscheidungszweigs durch einen *Sprungfehler* (①) gehört zu den augenfälligen Auswirkungen eines Kontrollflussfehlers. In diesem Fall ist die Evaluierung einer bedingten Anweisung beziehungsweise der daraufhin folgende Sprung fehlerhaft und führt so in einen falschen Zweig. Dieses Verhalten wird beispielsweise durch transiente Fehler in den für den Vergleich genutzten Registern oder Merkern verursacht. Im konkreten Beispiel ($x_c \neq y_c = z_c$) verzweigt die Bedingung in Zeile 3 ($x_c = y_c$) aufgrund eines transienten Fehlers fälschlicherweise in den `if`-Zweig. Im weiteren Verlauf wird der Vergleich in Zeile 4 zu der Annahme führen, dass z_c den Abweichler darstellt. Entsprechend wird die fehlerhafte Variante x_c als Gewinner ausgewählt sowie die Berechnung der dynamischen Signatur in Zeile 8 korrekt durchgeführt. Das Fehlverhalten des Mehrheitsentscheiders wird bei der nachfolgenden Decodierung des vermeintlichen Gewinners entlarvt, da das Ergebnis der dynamischen Signaturberechnung von der konstanten Signatur B_E abweichen wird – gemäß Gleichung 3.5 gilt an dieser Stelle $x_c - y_c \neq B_x - B_y$.

In der gleichen Art und Weise lassen sich auch beliebige *Sequenzfehler* erkennen. Hierbei springt der Kontrollfluss durch eine Verfälschung des Programmzählers beziehungsweise von Sprungadressen beliebig. In der Folge wird nicht nur das Falsche, sondern potenziell auch noch fehlerhaft gerechnet. Im zweiten Beispiel (②) springt der Kontrollfluss zwischen zwei Entscheidungszweigen und führt so zu einer gänzlich inkonsistenten Ausführung. Die Auswahl des Gewinners und die Berechnung der dynamischen Signatur in Zeile 5 erfolgen noch korrekt. Anschließend wechselt der Kontrollfluss fälschlicherweise den Zweig, wodurch die Rückgabe des Gewinners und der statischen Signatur in Zeile 9 erfolgt. Obwohl durch die hier angenommene Übereinstimmung aller Varianten grundsätzlich kein abweichendes Ergebnis des Gewinners möglich ist, wird das tatsächliche Fehlverhalten des CoRed-Mehrheitsentscheiders auch in diesem Fall korrekt erkannt. Wiederum weichen die konstante und die dynamische Signatur voneinander ab.

3.6 Fehlertoleranz

Grundsätzlich ermöglichen die CoRed-Mehrheitsentscheider und die arithmetische Codierung der Datenflüsse lediglich die Erkennung von transienten Fehlern. Dieses Verhalten ist in bestimmten Anwendungsfällen ausreichend, beispielsweise für die Umsetzung einer *kontrollierten Abschaltung* (engl. *fail stop*) im Fehlerfall. In diesem Fall sind bereits zwei Replikate ausreichend. Ausgehend von der bisher betrachteten dreifach redundanten Ausführung ist jedoch auch ein fehlertolerantes Verhalten der übrigen Komponenten wünschenswert. Für die Eingangsreplikation ist die Fehlertoleranz sowohl bei der deterministischen als auch der nichtdeterministischen Replikation möglich, beziehungsweise durch die inhärente Redundanz bereits konzeptbedingt gegeben.

Für den CoRed-Mehrheitsentscheider lässt sich fehlertolerantes Verhalten vergleichsweise einfach durch eine Rückwärtskorrektur (vgl. Kapitel 2.1.3) erreichen. Da der Mehrheitsentscheider zustandslos und die Ausführungsreihenfolge sequentiell ist, erfordert dies lediglich die wiederholte Ausführung des Entscheiders. Hierfür stehen im Wesentlichen die in Abbildung 3.7 dargestellten drei Umsetzungsalternativen zur Verfügung. In der bislang diskutierten Realisierung (Abbildung 3.7, links) erfolgt die Fehlererkennung implizit bei der Decodierung des Gewinners

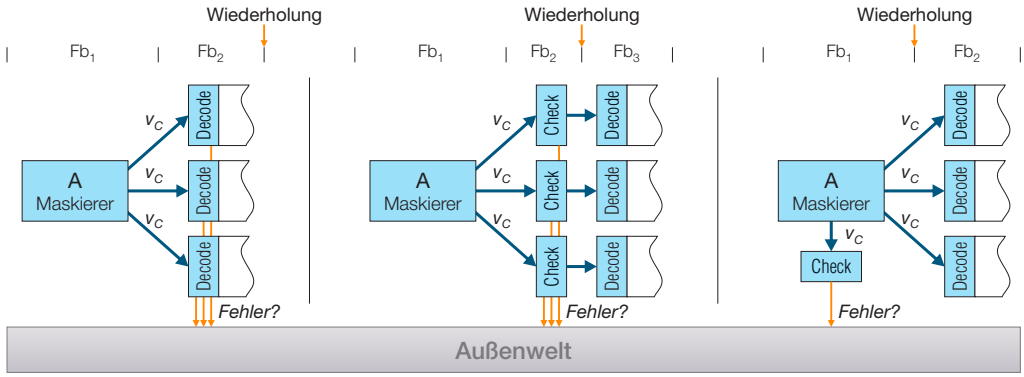


Abbildung 3.7: Fehlertolerante Auslegung des CoRed-Mehrheitsentscheiders.
Die Codierung von Datenflüssen und Mehrheitsentscheider ermöglicht zunächst lediglich Fehlererkennung. Fehlertoleranz lässt sich mittels Rückwärtskorrektur und damit der einfachen Wiederholung des zustandslosen Mehrheitsentscheiders erreichen. Im regulären Entwurf (links) erfolgt diese nach der Replikatausführung. Dies kann durch einen vorgeschalteten Akzeptanztest (`check()`) vermieden werden (Mitte). Durch die Einzelfehlerannahme kann der Test zu Optimierungszwecken auch in den Fehlerbereich des Maskierers wandern (rechts).

in den Replikaten. Im Fehlerfall wird der Mehrheitsentscheider für die Wiederholung vorgemerkt, welche jedoch erst nach dem Durchlauf der Replikate erfolgen kann. Dieses Vorgehen erfordert entsprechend die zusätzliche Wiederholung der Replikate selbst. Da diese üblicherweise zustandsbehaftet sind, müssen zusätzliche Rücksetzpunkte für die Replikate eingeführt werden. Dieses Problem lässt sich durch einen vorgeschalteten Überprüfungsschritt (Abbildung 3.6, Mitte) umgehen. In diesem Fall übernimmt ein Akzeptanztest (`check()`) die Fehlererkennung, wodurch die Wiederholung des Mehrheitsentscheiders vor der eigentlichen Decodierung und Replikatausführung erfolgen kann. Da die Replikate einen unterbrechungsfreien Durchlauf aufweisen ist in diesem Fall jedoch eine zeitliche Abfolge zwischen Überprüfung und Freigabe herzustellen, zum Beispiel durch Einführung einer zusätzlichen Prozessinkarnation für die Ausführung der Akzeptanztests. Aufgrund der Einzelfehlerannahme und unter Duldung von falsch positiven Resultaten (engl. *false positive*) lässt sich der Test jedoch in den Fehlerbereich des Mehrheitsentscheiders schieben. Die Überprüfung erfolgt hierbei auf einer vierten Kopie des Gewinners (Abbildung 3.7, rechts). Im Fehlerfall lässt sich der Mehrheitsentscheider direkt wiederholen.

3.7 Systemkomposition

In der Summe stellt CoRed eine zuverlässige Replikationsplattform zur Verfügung. Durch die Eliminierung der kritischen Fehlerstellen ist eine gesonderte Betrachtung dieser Stellen nicht länger notwendig. Entsprechend lässt sich die für die Anwendung sichtbare Fehlerrate gezielt reduzieren und damit die Zuverlässigkeit erhöhen. Für die Umsetzung ist lediglich eine redundante Ausführung der Anwendung notwendig. Für die Klasse der Regelungsanwendungen ist eine solche Replikation in der Regel einfach und transparent durchführbar. Über das Eingangsbeispiel aus Abbildung 3.3 hinaus, lassen sich dabei auch komplexere Anwendungen aus den CoRed-Strukturelemente zusammensetzen. Insbesondere der codierte Transfer des Ergebnisses erlaubt hierbei auch die Überprüfung entfernter Mehrheitsentscheider und damit die Korrektheit des entsprechenden Ausführungsstroms. Für eine selektive und einfache Umsetzung steht CoRed der Anwendung in Form einer Dienstschicht zur Verfügung.

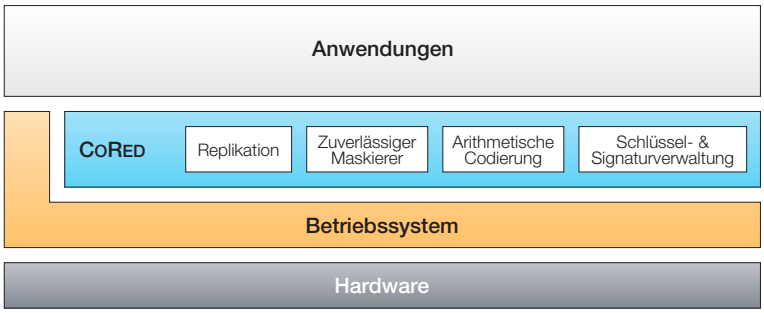


Abbildung 3.8: Übersicht über die CoRed-Dienstschicht.
Die von CoRed bereitgestellte Infrastrukturkomponenten und Mechanismen gliedern sich in Form einer *Dienstschicht* unterhalb der Anwendung auf der Ebene des Betriebssystems ein. Diese stellt die für die Replikation notwendigen Basisfunktionen wie Codierung oder Mehrheitsentscheider bereit. Hierzu zählen ebenfalls Stellvertreter für die Ein- und Ausgabe an gängige Systemdienste sowie gehärtete Funktionen und Treiber mit codierten Schnittstellen. Eine weitere Funktion der Dienstschicht ist die geeignete *Verwaltung der Codierungsschlüssel und Signaturen*.

3.7.1 CoRed-Dienstschicht

Die CoRed-*Dienstschicht* stellt die für die Absicherung einer sicherheitskritischen Anwendung notwendige Infrastruktur zur Verfügung. Sie gliedert sich dabei, wie in Abbildung 3.8 dargestellt, unterhalb der Anwendung auf der Ebene des Betriebssystems ein. Zu den zentralen Funktionen zählen hierbei die Unterstützung der *Replikation* durch die in Kapitel 3.4 eingeführten Strukturelemente für die Anbindung der Replikate an die Außenwelt. Darüber hinaus sind hier auch die anwendungsunabhängigen Stellvertreter beziehungsweise Akzeptanzmaskierer für die Emulation codierter Systemschnittstellen und Hardwaretreiber angesiedelt. Hierzu zählen insbesondere die für die *arithmetische Codierung* der Datenströme notwendigen Operationen für die Eingabe und Ausgabe. Nicht zuletzt stellt die CoRed-Dienstschicht die entscheidenden *zuverlässigen Mehrheitsentscheider* aus Kapitel 3.5 zur Verfügung. Über die in Kapitel 2.2.3 vorgestellten, vergleichbaren Replikationsschichten auf Betriebssystemebene hinaus, eliminiert CoRed damit nicht nur die kritischen Fehlerstellen der softwarebasierten Replikation, sondern erweitert auch die Möglichkeiten zur Systemkomposition durch die Bereitstellung der codierten Schnittstellen.

Ein weiterer wichtiger Aspekt für die Umsetzung der codierten Datenströme und des CoRed-Mehrheitsentscheiders ist die Verwaltung der *Codierungsschlüssel und Signaturen*. Diese werden für jede Anwendung und jeden Ausführungsstrang separat geführt und eineindeutig an die codierten Datenströme vergeben. Für die korrekte Funktion des CoRed-Mehrheitsentscheiders müssen sich beispielsweise die Signaturen (B) für die drei Varianten x_c, y_c und z_c unterscheiden. Die Auswahl geeigneter Schlüssel und Signaturen ist Gegenstand von Kapitel 4.

3.7.2 Abbildung auf Aufgaben des Echtzeitsystems

Als letzter Schritt nach der Absicherung der Anwendung, bleibt die Abbildung der entstandenen Komponenten auf das Echtzeitbetriebssystem. Die betrachtete Klasse der sicherheitskritischen Regelungsanwendungen weist ein hohes Maß an Regelmäßigkeit auf, da die Regelungsaufgabe üblicherweise mit einer festen Periode ausgeführt wird [FPW97, ÅW11]. Letztlich lässt sich die Anwendung damit ausschließlich auf periodische *Aufgaben* (engl. *task*) des Echtzeitsystems abbilden. Eine Aufgabe definiert sich hierbei durch Periode, maximale Ausführungszeit (engl. *worst case execution time*), Termin (engl. *deadline*) und Phase [Liu00]. Die typischen Aufgaben der Regelungsanwendung sind *abtasten*, *regeln* und *steuern*. Die auslösenden Ereignisse ergeben sich damit letztlich aus den vorab bekannten Regelperioden.

Auswirkungen der Replikation

Durch die Replikation kommen in der Folge weitere Aufgaben in Form von zusätzlichen Replikaten, Replikatoren und Mehrheitsentscheidern hinzu. Die zeitlichen Domänen der entstehenden Aufgabenmenge sind jedoch identisch, weshalb sich diese problemlos miteinander verschmelzen lassen. Folglich bleiben sowohl die auslösenden Ereignisse, Perioden und Termine bestehen. Lediglich die maximale Ausführungszeit erhöht sich entsprechend um die Anzahl der Replikate. Darüber hinaus ergeben sich jedoch zusätzliche Abhängigkeiten in Form der Produzenten/Konsumenten-Verhältnisse zwischen den Mehrheitsentscheidern und den Replikaten, deren Varianten nun konsumierbare Betriebsmittel darstellen. Unabhängig von der konkreten Realisierung kann der Datenfluss zwischen den einzelnen Teilkomponenten entsprechend als Nachrichtenaustausch aufgefasst werden.

Koordinierung

Die einzelnen Arbeitsaufträge bilden entsprechend einen gerichteten Abhängigkeitsgraphen, welcher deren Beziehungen untereinander abbildet [Liu00]: Ein Mehrheitsentscheid kann beispielsweise erst nach der Berechnung der zugeordneten Replikate und der Bereitstellung der Ergebnisse durchgeführt werden. Durch die identischen Aktivierungsperioden und das damit verbundene hohe Maß an sequentieller Kopplung ist eine statische Serialisierung der Arbeitsaufträge jedoch problemlos möglich.

Zusammen mit dem umfassenden a priori Wissen über die periodischen Aufgaben und der konzeptbedingten Abwesenheit von komplexen Arbeitsaufträgen ist somit eine konstruktive Koordinierung durch eine statische Einplanung und einen geeigneten Phasenversatz möglich und erstrebenswert. Eine aufwendige Synchronisierung der Arbeitsaufträge, zum Beispiel durch einen Signalaustausch zwischen den Replikaten und den Mehrheitsentscheidern, kann daher entfallen. Das resultierende Aufgabensystem ist entsprechend strikt periodisch und kann beispielsweise durch eine Ablauftabelle und eine getaktete Ausführung realisiert werden.

3.8 Zusammenfassung

Die erweiterte Problemanalyse dieses Kapitels zeigt anschaulich, wie die praktische Umsetzbarkeit der softwarebasierten Replikation unter den Zuverlässigkeitsengpässen und kritischen Fehlerstellen der redundanten Ausführung leidet. Der kombinierte Redundanzansatz CORED löst dieses Problem durch die namensgebende Kombination von softwarebasierter Replikation auf Ebene des Betriebssystems und einer auf ANBD-Codes basierenden arithmetischen Codierung der verbleibenden kritischen Fehlerstellen.

Die Codierung der Datenströme in Kombination mit der Replikation der Eingänge beziehungsweise der Konsolidierung der Ausgänge erlaubt zunächst die Erweiterung der Fehlererfassung bis an die äußersten Grenzen des in Software realisierbaren Redundanzbereichs. Der Einsatz codierter Eingänge beziehungsweise Ausgänge ermöglicht zudem die Komposition komplexer Anwendungen und darüber hinaus eine Erweiterung des Redundanzbereiches über die Grenzen des Rechensystems hinaus. Die Weitergabe eines codierten Stellwertes erlaubt beispielsweise die nachgelagerte Überprüfung der Korrektheit der replizierten Ausführung und des Mehrheitsentscheids bis kurz vor der endgültigen Verwendung des Wertes durch den Aktor.

Das Schlüsselelement des Ansatzes sind die zuverlässigen Mehrheitsentscheider, welche eine Entscheidung unter der Beibehaltung der arithmetischen Codierung der Varianten ermöglichen und darüber hinaus eine effektive Kontrollflussüberwachung auf Basis der Codierungssignaturen gestatten. CORED entwickelt hierfür einen maßgeschneiderten Entwurf des Mehrheitsentscheiders

als codierte Operation $\textcircled{v}_c$, welche sich nahtlos in die bestehende Codierung einfügen lässt. Dies erlaubt die sichere Überprüfung des Ergebnisses im Zuge der nachfolgenden Verwendung. Die CoRed-Mehrheitsentscheider stellen letztlich die erforderlichen *Wächter über die Wächter* dar und ermöglichen eine Fehlererkennung in der Replikationsinfratruktur und damit letztlich die vollständige Beseitigung der Zuverlässigkeitsengpässe.

Aus Sicht des Anwendungsentwicklers fügt sich CoRed als Dienstschicht auf Ebene des Betriebssystems ein. Ausgehend von einem replizierbaren Prozesssystem mit definierten Eingängen und Ausgängen sind dabei keine weiteren Schritte notwendig. Die arithmetische Codierung der Datenflüsse wie auch die hierfür notwendige Verwaltung der Codierungsschlüssel und Signaturen sind ebenso Teil der Dienstschicht und die möglicherweise notwendige Emulation der Systemschnittstellen und Treiber. Durch die Eliminierung der kritischen Fehlerstellen ermöglicht CoRed die gezielte Beeinflussung der Zuverlässigkeit der zu schützenden Anwendung. Dabei reduziert sich die Zuverlässigkeitsbetrachtung auf die gängigen Verfahren für die Replikation wie sie in Kapitel 3.1.2 vorgestellt wurden.

4

Von der Theorie in die Praxis

Effektive Fehlererkennung – Implementierung und Evaluation

Der Entwurf des CoRed-Ansatzes im vorangegangenen Kapitel ermöglicht einen durchgehenden Schutz sicherheitskritischer Anwendungen vor transienten Fehlern und eine konzeptionell *vollständige Fehlererfassung* im Sinne des Fehlermodells. Hierfür ist vor allem die geeignete Instrumentalisierung der Signaturen (*BD*-Anteil) durch die arithmetische Codierung innerhalb des Mehrheitsentscheiders ausschlaggebend. Mit Blick auf die existierenden Ansätze und die auf arithmetischer Codierung basierenden Fehlertoleranztechniken (vgl. Kapitel 2.2.2) zeigt sich jedoch, dass trotz einer theoretisch vollständigen Fehlererfassung der Codierung weiterhin unerkannte Datenfehler in signifikantem Umfang auftreten. Die Ursachen hierfür lassen sich im Einzellfall nur schwer differenzieren. Im Allgemeinen kommen sowohl Lücken und Umsetzungsfehler in der technischen Realisierung als auch Schwächen in der Erkennungsleistung der arithmetischen Codierung infrage. Anschaulich lässt sich diese Problematik mit der Systemsicherheit (engl. *security*) vergleichen: Die Vermeidung von Sicherheitslücken (Schwachstellen) schützt nicht vor einem Einbruch durch ein schlecht gewähltes Passwort. Umgekehrt hilft ein perfektes Passwort ebenso wenig gegen vorhandene Sicherheitslücken. In diesem Kapitel verschiebt sich der Fokus daher auf die praktische Umsetzung des CoRed-Ansatzes und den Schritt von der *Theorie in die Praxis*. Der maßgebliche Baustein für eine gleichermaßen *zuverlässige Fehlererkennung* ist hierbei der *AN*-Anteil der Codierung.

Im Gegensatz zu dem vorangegangenen Beispiel aus der Systemsicherheit, lässt sich die Auswahl geeigneter Codierungsschlüssel nicht vollständig losgelöst von der technischen Realisierung des CoRed-Ansatzes betrachten. Der Aufbau dieses Kapitels folgt daher nicht dem kanonischen Verlauf von Implementierung zu Evaluation. Stattdessen werden die verschiedenen Implementierungsebenen schrittweise beleuchtet und evaluiert. Die Betrachtung beginnt auf der Ebene der mathematischen *Codierungstheorie* (engl. *coding theory*) [Ham80] mit der Auswahl der Codierungsparameter in Kapitel 4.2 und setzt sich über die Fallstricke bei der binären Darstellung der Codierung bis auf die Befehlssatzebene der Hardware in Kapitel 4.3 fort. In jedem Schritt kommen dabei jeweils angepasste Methoden für die Evaluation der Effektivität zum Einsatz. Diese reichen von der rechnerischen Bestimmung der Erkennungsleistung, über deren Messung durch

Fehlersimulation bis hin zu der Injektion von Bitfehlern in den Hardwarekontext der Anwendung. Das Ziel ist letztlich der abschließende Nachweis der Übereinstimmung von Konzept und Realität, sowohl für die technische Umsetzung als auch für die Codierung an sich. Darüber hinaus diskutiert Kapitel 4.4 die Evaluationsergebnisse und deren Verallgemeinerbarkeit beziehungsweise die unvermeidbaren Einschränkungen.

4.1 Erweiterte Fehlerhypothese

Bislang definieren das Fehlermodell in Kapitel 2.1.1 sowie die grundlegende Fehlerhypothese in Kapitel 3.2.1 die *Art* der auftretenden transienten Fehler beziehungsweise deren *Häufigkeit* im Sinne der einzelnen Fehlerbereiche. Der CoRed-Entwurf bietet in diesem Zusammenhang konzeptionell eine *vollständige Fehlererfassung*. Dessen ungeachtet unterscheiden sich transiente Fehler jedoch auch in ihrem *Schweregrad*. Dieser gibt letztlich an wie stark ein Strukturelement der Software verfälscht wurde. Der Inhalt eines Registers kann beispielsweise durch eine Vielzahl von verschiedenen *Fehlermustern* (engl. *fault pattern*) betroffen sein, von einem einfachen Einzelbitfehler bis hin zu einer kompletten Inversion des gesamten Registerinhalts. Entsprechende Messungen zeigen hierbei eine klare Häufung minder schwerer Fehlermuster. Maiz et al. [MHZA03] ermitteln in praktischen Experimenten für 90 und 130 nm Strukturbreiten in der Summe eine über 95-prozentige Wahrscheinlichkeit für Einzelbitfehler. Entsprechend führen lediglich unter fünf Prozent der transienten Fehler zu Mehrfachbitfehlern mit maximal fünf verfälschten Bits. Diese Ergebnisse stehen in Übereinstimmung mit weiteren Untersuchungen [MW79, KKA95, LSHC07], unter anderem auch einer breit angelegten Studie von Nightingale et al. [NDO11] zu den durch Hardware verursachten Absturzursachen in Arbeitsplatzrechnern.

Die dem CoRed-Ansatz zugrundeliegende Fehlerhypothese erweitert sich entsprechend um die Schwere der auftretenden transienten Fehler und die *zuverlässige Fehlererkennung* von zumindest Einzelbitfehlern. Für die Fehlerbereiche gilt entsprechend die aufgrund der vorliegenden Daten häufig eingesetzte *Einzelbit-Einzelfehler-Annahme* (engl. *single-bit single-error assumption*).

Der Schweregrad transienter Fehler

Neben der *Art* (Fehlermodell) und *Häufigkeit* (Fehlerbereich) der auftretenden transienten Fehler schließt die erweiterte Fehlerhypothese auch den *Schweregrad* (Bitfehlermuster) mit ein. Diese geht von der häufig genutzten *Einzelbit-Einzelfehler-Annahme* aus, da sich über 95 % der transienten Fehler als Einzelbitfehler manifestieren [MHZA03].

4.2 Wahl der Codierungsparameter

Die Realisierung einer zuverlässigen Fehlererkennung im Sinne des Schweregrads transienter Fehler ist im Falle der Codierung letztlich keine alleinige Frage der Codierungsvorschrift beziehungsweise des Entwurfs der Fehlertoleranzmaßnahme. Vielmehr entscheidet der Umfang der Redundanz über die Effektivität und Fehlererkennungsleistung. Bei der arithmetischen Codierung ist hierfür vor allem die geeignete Auswahl der Codierungsparameter, bestehend aus Codierungsschlüssel (A) und der eingesetzten Signaturen (B) ausschlaggebend. Der folgende Abschnitt zeigt daher detailliert, wie sich die entsprechenden Parameter auswählen lassen und welchen Einfluss die binäre Darstellung der Codierung auf die erreichbare Zuverlässigkeit des Codes hat.

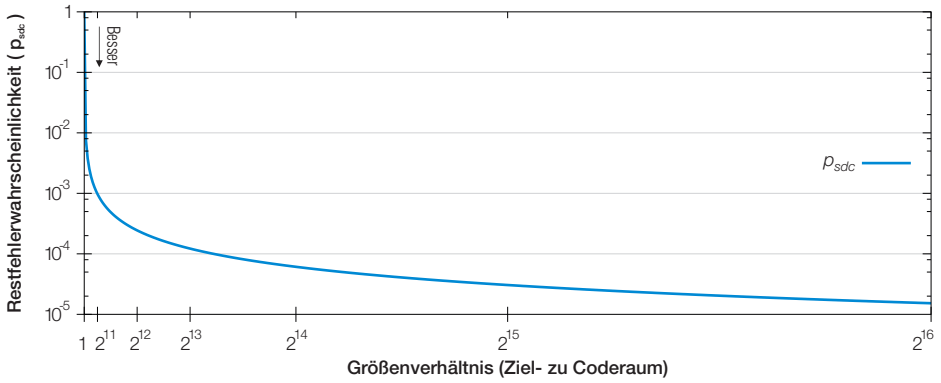


Abbildung 4.1: Die Restfehlerwahrscheinlichkeit im Verhältnis zur Informationsredundanz.
Trotz Codierung können transiente Fehler zu einer Mutation zwischen gültigen Codewörtern und damit zu unerkannten Datenfehlern führen. Diese *Restfehlerwahrscheinlichkeit* (p_{sdc}) hängt im Allgemeinen vom Umfang der hinzugefügten Informationsredundanz ab (vgl. Gleichung 4.1). Sie entspricht eins, falls die Anzahl der Codewörter gleich der Anzahl der Nutzwörter ist. Mit zunehmender Redundanz und Coderaumgröße nimmt die Wahrscheinlichkeit entsprechend ab.

4.2.1 Restfehlerwahrscheinlichkeit der Codierung

Die Fehlererkennung der Codierung basiert, wie bereits in Kapitel 2.2.2 erläutert, auf der dünneren Besetzung des durch die Codierungsvorschrift erzeugten *Coderaums* (engl. *code space*) und dem hierdurch entstehenden Abstand zwischen den enthaltenen Codewörtern. Vereinfacht dargestellt transformiert die Codierung Datenwörter (n Bit) durch Hinzufügen von k Bit redundanter Information in Codewörter ($n+k$ Bit). Die Fehlerdiagnose erfolgt im Umkehrschluss durch einen Absoluttest im Zuge der Rücktransformation des Codewortes.

Ungeachtet einer theoretisch vollständigen Fehlererfassung, wie sie beispielsweise bei den ANBD-Codes gegeben ist, sind unerkannte Datenfehler trotz der eingesetzten Codierung nicht völlig auszuschließen. Die hierbei verbleibende *Restfehlerwahrscheinlichkeit* (engl. *residual error probability*) (p_{sdc}) für die Mutation von einem gültigen Codewort in ein anderes gültiges Codewort ergibt sich letztlich aus dem Umfang der hinzugefügten Informationsredundanz [Fro77]:

$$p_{sdc} = \frac{\text{Anzahl gültiger Codeworte} - 1}{\text{Anzahl möglicher Codeworte} - 1} = \frac{2^n - 1}{2^{n+k} - 1} \xrightarrow[n\to\infty]{} \frac{1}{2^k} \Leftrightarrow \frac{1}{A} \tag{4.1}$$

Der Wert von p_{sdc} nimmt mit wachsendem Verhältnis zwischen gültigen (2^n) und möglichen Codewörtern (2^{n+k}) exponentiell ab. Die Effektivität der Fehlererkennung ist entsprechend proportional zum Umfang der eingesetzten Informationsredundanz. Im Falle der arithmetischen Codierung entspricht die Zahl der möglichen Codewörter $A \cdot 2^n$, weshalb p_{sdc} in diesem Fall $1/A$ entspricht. Abbildung 4.1 veranschaulicht exemplarisch, wie die Restfehlerwahrscheinlichkeit mit der Größe des Codierungsschlüssels A abnimmt. Diese theoretische Abschätzung geht insgesamt davon aus, dass beliebige Verfälschungen eines Codewortes mit der gleichen Wahrscheinlichkeit auftreten und vollkommen unabhängig von der Art und Schwere der transienten Fehler beziehungsweise der verursachten Anzahl der Bitfehler sind. Da dies keine weiteren Annahmen hinsichtlich der Fehlerhypothese erfordert, eignet sich die Restfehlerwahrscheinlichkeit insbesondere als *untere Schranke* für die Zuverlässigkeitsbetrachtung der eingesetzten Codierung. Der Wert von p_{sdc} spielt daher insgesamt eine wichtige Rolle für die theoretische und praktische Bewertung von auf Informationsredundanz basierender Fehlertoleranz [Ech90].

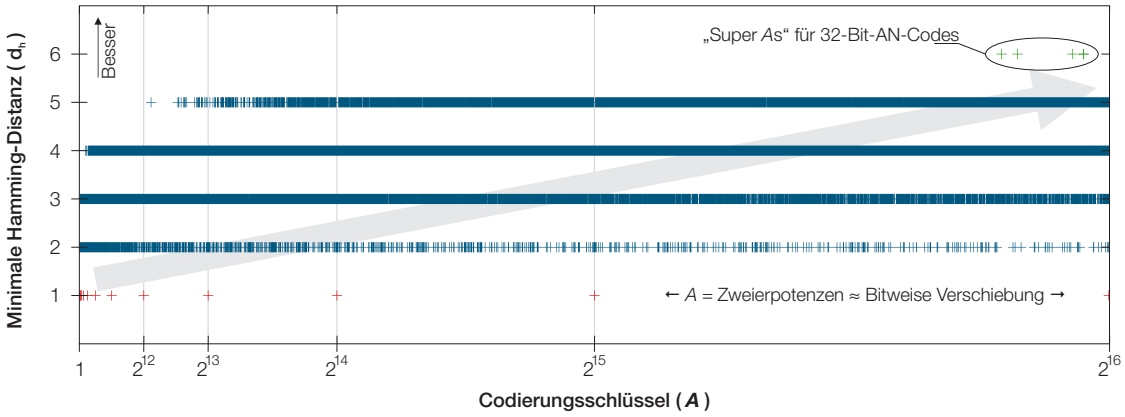

Minimale Hamming-Distanz (d_h)
Besser
„Super As“ für 32-Bit-AN-Codes
← A = Zweierpotenzen ≈ Bitweise Verschiebung →
Codierungsschlüssel (A)

Abbildung 4.2: Minimale Hamming-Distanz für einen 32-Bit-Code mit 16-Bit-Codierungsschlüsseln.
Die Effektivität der Codierung steigt erwartungsgemäß mit der Schlüsselgröße (A) an. Auffällig ist die breite Streuung der Ergebnisse, was allgemeingültige Aussagen für die Auswahl der Codierungsschlüssel nur schwer zulässt. Ausreißer nach unten stellen die Zweierpotenzen dar, sie entsprechen einer bitweisen Verschiebung und ermöglichen keine sichere Fehlererkennung. Demgegenüber weisen die *Super-As* eine Distanz von sechs auf und erlauben die Erkennung von 5-fach-Bitfehlern.

4.2.2 Auswahl der Codierungsschlüssel

Mit Blick auf die Abschätzung der Restfehlerwahrscheinlichkeit erscheint die Auswahl großer Werte für A als Codierungsschlüssel sinnvoll, frei nach dem Motto: *Je größer, desto besser*.

Primzahlen als Codierungsschlüssel?

In der Literatur existieren darüber hinaus wenig Hinweise hinsichtlich der geeigneten Auswahl von A. Insbesondere für die ANBD-Codierung wird verschiedentlich der Einsatz von *Primzahlen* favorisiert, vor allem um die Anzahl der möglichen Faktoren zwischen verschiedenen Verarbeitungsströmen zu reduzieren [For89, Oze92, Sch11b]. Dieses von einem mathematischen Standpunkt aus intuitiv plausibel erscheinende Vorgehen bleibt im Grundsatz jedoch weitgehend unbegründet. In ihrer experimentellen Untersuchung der Codierungseigenschaften verschiedener Schlüssel, identifiziert Schiffel [Sch11b] demgegenüber auch einige zusammengesetzte Zahlen, welche eine ähnliche oder sogar bessere Fehlererkennungsleistung aufweisen als vergleichbare Primzahlen. Wiederum lassen sich aus dieser Analyse jedoch keine eindeutigen Auswahlkriterien für die Codierungsschlüssel ableiten, weshalb Schiffel ebenfalls zu großen Primzahlen rät.

Experimentelle Bestimmung der Effektivität mittels Hamming-Distanz

Weder die Restfehlerwahrscheinlichkeit als untere Schranke für die Effektivität eines Codierungsschlüssels noch der Verweis auf große Primzahlen erscheint ein zielführendes Auswahlkriterium darzustellen. Von einer binären Betrachtung des Codes ausgehend, ist der einzige Sinn und Zweck des Schlüssels (A) die Generierung von *robusten* Bitmustern. Die durch die Codierung erzeugten Codewörter sollen entsprechend ein hohes Maß an Unempfindlichkeit gegenüber Bitfehlern aufweisen. Eine allgemeine Kenngröße der Codierungstheorie für diese Art der Robustheit ist die *Hamming-Distanz* (engl. *hamming distance*) (d_h) [Ham50]. Diese gibt an, in wie vielen Bitpositionen sich zwei Codewörter unterscheiden. Die *Hamming-Minimaldistanz* (engl. *minimum Hamming distance*) eines Codes entspricht folglich dem kleinsten Abstand aller Codewörter untereinander. Die durch einen Code sicher erkennbare Zahl an Fehlern (f) entspricht $f = d_h - 1$

und lässt sich somit direkt aus der Hamming-Distanz ableiten. Eine ausführliche Erläuterung des zugrunde liegenden Konzepts gibt der Exkurs auf Seite 80.

Die Bestimmung der Hamming-Minimaldistanz ermöglicht folglich eine direkte Bewertung der Effektivität eines Codes beziehungsweise des eingesetzten Codierungsschlüssels (A) und erscheint daher insbesondere geeignet für dessen Auswahl. Die Berechnung der Hamming-Minimaldistanz erfordert nunmehr den systematischen Vergleich aller Codewörter jedes Codes. Der Berechnungsaufwand steigt dabei quadratisch mit dem Wertebereich der Eingabedaten an. Die in diesem Fall angenommene Gleichverteilung in 16-Bit-Werte (v) und 16-Bit-Schlüssel (A) führte somit zu über $1{,}4 \cdot 10^{14}$ Einzelexperimenten für die Berechnung aller hierdurch darstellbaren 32-Bit-AN-Codes. Abbildung 4.2 zeigt die Ergebnisse der experimentellen Bestimmung der Hamming-Minimaldistanz für alle 16 Bit Codierungsschlüssel. Die sich ergebenden Werte für d_h reichen in Abhängigkeit vom gewählten A von eins bis sechs. Entsprechend lassen sich durch die jeweiligen Codes bis zu fünf Bitfehler sicher erkennen. Eine Ausnahme bilden in diesem Zusammenhang die Zweierpotenzen ($A = 2^k$). Da es sich hierbei lediglich um eine bitweise Verschiebung des Datenwortes handelt, ist eine sichere Fehlererkennung generell nicht gegeben. Obwohl die erwartete Zunahme der Erkennungsleistung mit ansteigendem A anhand der Verlagerung der Messpunkte in Abbildung 4.2 grundsätzlich zu erkennen ist, variiert die gemessene Minimaldistanz zwischen benachbarten Werten teilweise *erheblich*. Die Auswirkungen dieses Verhaltens lassen sich am Beispiel der folgenden drei, vergleichsweise großen, Codierungsschlüssel veranschaulichen:

$A = 58\,368$	$d_h = 2$	Erkennbare Bitfehler $= 1$
$= 58\,831$ (Primzahl)	$= 3$	$= 2$
$= 58\,659$	$= \mathbf{6}$	$= \mathbf{5}$

Demnach weist der zusammengesetzte Schlüssel 58 659 eine doppelt so große Hamming-Distanz als die benachbarte Primzahl 58 831 auf. Dieses auf den ersten Blick unverständliche Verhalten begründet sich in der nicht-systematischen Darstellung der ANBD-Codes (vgl. Kapitel 2.2.2), in welcher die angenommenen n Datenbits und k Prüfbits untrennbar zusammen gespeichert und verarbeitet werden. Entsprechend leitet sich der entstehende Abstand nicht notwendigerweise aus den für die Darstellung von A genutzten k Bits ab, sondern vielmehr aus der binären Darstellung des Codewortes in Form von $A \cdot v$. Das Produkt der beiden Werte entscheidet damit letztlich über die Robustheit der entstehenden Bitmuster. In der Folge ist *je größer, desto besser* eine *irreführende Annahme* für die Auswahl eines geeigneten Codierungsschlüssels.

Den passenden Schlüssel finden

Die Effektivität der arithmetischen Codierung hängt in hohem Maße von dem gewählten Codierungsschlüssel ab. Über die in der Literatur empfohlenen Kriterien hinaus, erlaubt jedoch erst die experimentelle Bestimmung der Hamming-Minimaldistanz eine fundierte Auswahl. Für die untersuchten 32-Bit-Codes konnten Schlüssel für die zuverlässige Erkennung von bis zu 5-fach-Bitfehler gefunden werden.

Die besten Codierungsschlüssel

Nichtsdestotrotz befinden sich die gemäß der Hamming-Distanz leistungsfähigsten Schlüssel am oberen Ende des Wertebereichs. Diese im Folgenden als *Super-As* bezeichneten fünf Codierungsschlüssel 58 659, 59 665, 63 157, 63 859 und 63 877 – keine davon eine Primzahl – weisen eine

Exkurs: Die Hamming-Distanz

Basierend auf der mathematischen Betrachtung der Restfehlerwahrscheinlichkeit (p_{sdc}) erscheint eine vollständige Fehlererfassung zunächst ausgeschlossen, da p_{sdc} in Näherung $1/A$ entspricht. Im Falle von $k = 3$ Prüfbits, entspricht dies einer Wahrscheinlichkeit von annähernd 12,5 % für das Versagen der Fehlererkennung und das Auftreten eines unerkannten Datenfehlers.

Binäre Betrachtung: Nicht alle Fehler sind gleich wahrscheinlich

Da sich transiente Fehler in Form von Bitfehlern manifestieren, unterscheiden sich diese Wahrscheinlichkeiten jedoch erheblich aufgrund der Binärdarstellung der Codewörter. So lässt sich der Wert $X = 128$ $(\texttt{10000000})_2$ durch einen Bitfehler zu $X_{\text{fehlerhaft}} = 129$ $(\texttt{10000001})_2$ verfälschen, eine Mutation zum $X_{\text{fehlerhaft}} = 127$ $(\texttt{01111111})_2$ benötigt demgegenüber jedoch acht einzelne Bitfehler. Zur Charakterisierung des Abstands im Binärsystem wird daher die *Hamming-Distanz* (engl. *Hamming distance*) [Ham50] (auch *Hamming-Abstand*) genutzt. Diese beschreibt die Anzahl der Bits, in denen sich zwei Wörter voneinander unterscheiden – in diesem Beispiel ist der Abstand demnach eins beziehungsweise acht. Darauf aufbauend entspricht die *Hamming-Minimaldistanz* (engl. *minimum Hamming distance*) d_h eines Codes dem Minimum aller Abstände zwischen zwei beliebigen, gültigen Codewörtern. Für die Menge der Codewörter $\{\texttt{00000, 00111, 11100, 11011}\}_2$ ist die Minimaldistanz entsprechend $d_h = 3$.

Hamming-Minimaldistanz (d_h)		**1**	**2**	**3**	**3**	**4**	**4**	**5**	**5**	**5**
Maximalfehlerzahl (f)	Erkennbare Bitfehler ($d_h = f + 1$)	0	1	— 2 —		— 3 —		—— 4 ——		
	Korrigierbare Bitfehler ($d_h = 2f + 1$)	0	0	— 1 —		— 1 —		—— 2 ——		
Gültiges Fehlermodell ($d_h = \alpha + \beta + 1$)	Erkennbare Bitfehler (α)	0	1	2	1	3	2	4	3	2
	Korrigierbare Bitfehler (β)	0	0	0	1	0	1	0	1	2

Tabelle 4.1: Maximalfehlerzahl und gültige Fehlermodelle im Verhältnis zur Hamming-Minimaldistanz.
Die Minimaldistanz (d_h) eines Codes bestimmt die Zahl der durch diesen *sicher* erkennbaren ($f + 1$) beziehungsweise korrigierbaren ($2f + 1$) Bitfehler (f). *Abbildung in Anlehnung an [Muk08].*

Zuverlässige Fehlererkennung

Die minimale Hamming-Distanz bestimmt, wie viele Fehler f durch einen bestimmten Code *zuverlässig* erkannt beziehungsweise korrigiert werden können. Eine sichere Fehlererkennung ist für $\geq f+1$ gewährleistet, da kein Codewort durch f fehlerhafte Bits in ein anderes gültiges Codewort überführt werden kann. Darüber hinaus muss der Mindestabstand $d_h \geq 2f+1$ betragen, um fehlerhafte Codewörter korrigieren zu können. Die Rückführung basiert wiederum auf dem Abstand zu anderen gültigen Codewörtern, welcher in jedem Fall mindestens $f+1$ entspricht. Der ursprüngliche Wert des Codewortes lässt sich daher eindeutig über den kleinsten Abstand ermitteln. Da die Fehlerkorrektur einen doppelt so großen Hamming-Abstand erfordert, lassen sich entsprechend mehr Bitfehler erkennen als korrigieren. Die Erkennungsleistung hängt hierbei allerdings von der Zahl der zu erkennenden Fehler ab. Ausgehend von den beiden Codewörtern $X = \texttt{0011}_2$ und $Y = \texttt{1100}_2$ mit $d_h = 4$, ist eine Korrektur von einem und die Erkennung von zwei Bitfehlern umsetzbar. 3-fach-Bitfehler ließen sich zwar grundsätzlich erkennen, eine zuverlässige Unterscheidung zwischen diesen und Einzelbitfehlern ist an diesem Punkt jedoch nicht mehr möglich (z. B., $X_{\text{f}=1} = \texttt{0111}_2 = Y_{\text{f}=3}$). In diesem Beispiel würde die Korrektur des fehlerhaften Codewortes $\texttt{0111}_2$ durch den geringeren Abstand stets zu X, und damit im Fall des 3-fach-Bitfehlers $Y_{\text{f}=3}$ zu einem unerkannten Datenfehler führen. Tabelle 4.1 zeigt exemplarisch die für ein Fehlermodell aus α erkennbaren und β korrigierbaren Fehlern gültigen Kombinationen. Die notwendige Hamming-Minimaldistanz bestimmt sich entsprechend durch $h = \alpha + \beta + 1$, wobei $\alpha \leq \beta$.

Distanz von sechs auf und ermöglichen somit die zuverlässige Erkennung von 5-fach-Bitfehlern. Ausgehend von der von Maiz et al. [MHZA03] ermittelten Maximalfehlerzahl von fünf, ermöglichen diese Codierungsschlüssel eine ausnahmslose und vollständige Fehlererkennung aller transienter Fehler. Da CoRed die codierten Verarbeitungsströme räumlich voneinander isoliert, spielen gemeinsame Primfaktoren für die Fehlererkennungsleistung keine Rolle. Darüber hinaus weist die Klasse der Primzahlen eine mit den zusammengesetzten Zahlen vergleichbare Streuung der Hamming-Minimaldistanz auf. Entsprechend bringt in diesem speziellen Anwendungsfall der Einsatz von Primzahlen keinen ersichtlichen Vorteil.

4.2.3 Auswahl der Signaturen

Grundsätzlich basiert die Fehlererkennung der arithmetischen Codierung auf dem Codierungsschlüssel (A). Die eingesetzten Signaturen (B) dienen lediglich dazu, neben den reinen Datenfehlern auch die anderen Fehlerklassen für die Decodierung sichtbar zu machen. Eine Signatur entspricht damit letztlich einer gezielten Verfälschung des Codewortes durch die Einbringung eines *Rests* in den Basiscode ($A \cdot v$). Da die Signaturen additiv auf das Codewort angewendet werden, entsprechen sie lediglich einer Verschiebung des Codewortes innerhalb des Wertebereichs. Sie wirken sich daher nicht störend auf die Auswahl eines geeigneten Codierungsschlüssels (A) aus. Erweiterte Experimente zur Bestimmung der Hamming-Minimaldistanz unter Berücksichtigung der Signatur zeigten entsprechend auch keinen Einfluss von B auf die sich ergebenden Werte von d_h. Dessen ungeachtet ist eine sorgfältige Auswahl der Signaturen unumgänglich, um eine Verfälschung einer gültigen Signatur in eine andere gültige Signatur zu verhindern.

In der Literatur finden sich wiederum keine exakten Angaben über die geeignete Wahl der Signaturen. Neben der notwendigen Forderung, dass $0 < B < A$, beschränken sich diese auf eine angemessene Verteilung der Signaturwerte. Schiffel [Sch11b] empfiehlt darüber hinaus die dynamische Anpassung der Signaturen im Verlauf längerer codierter Berechnungen, um eine zufällige Übereinstimmung der Signaturen bei Zwischenergebnissen zu vermeiden. Das Ergebnis einer codierten Addition von X_c und Y_c kann beispielsweise zu der Signatur von Z_c führen, falls $B_z = B_x + B_y$. In diesem Fall lässt sich das Ergebnis der Addition nicht mehr von Z_c unterscheiden. Ein etwaiger Adressierungsfehler ist somit nicht mehr zuverlässig erkennbar. Im Falle von CoRed lassen sich die Signaturen und Zwischenergebnisse für alle Verarbeitungsströme jedoch im Vorfeld exakt bestimmen und geeignet parametrieren. Da diese zudem strikt voneinander isoliert ablaufen, sind ungewollte Wechselwirkungen zwischen den Signaturen verschiedener CoRed-Komponenten und somit auch unerkannte Adressierungsfehler ausgeschlossen.

Um die differenzierenden Eigenschaften der Signaturen auch in der Gegenwart von transienten Fehlern sicherzustellen, fasst CoRed die Menge der notwendigen Signaturen daher wiederum als Code auf und nutzt entsprechend die Hamming-Minimaldistanz zu dessen Optimierung. Für die Signaturen des CoRed-Mehrheitsentscheiders (vgl. Kapitel 3.5) ergeben sich daher folgende Auswahlschritte. Zunächst leitet sich die zu betrachtende Menge der Signaturen aus den drei Eingabevarianten (B_x, B_y und B_z) sowie dem Gewinner (B_{dyn}) ab. Dessen Signatur weist vier mögliche Werte für den Ausgang des Entscheids auf und setzt sich aus den Signaturen der Varianten zusammen. Um negative Werte zu verhindern, müssen die Signaturen der Varianten zunächst die folgende Bedingung erfüllen: $B_x < B_y < B_z$. Darüber hinaus werden die drei Eingangssignaturen derart gewählt, dass die Hamming-Minimaldistanz der insgesamt genutzten sieben Signaturen dem Abstand des gewählten Codierungsschlüssels entspricht. Es gelten daher einheitlich die auf der Hamming-Distanz basierenden Auswahlkriterien, sowohl für die Schlüssel, als auch für die Signaturen. Für die zuvor genannten Super-*A*s mit $d_h = 6$ erfüllen zum Beispiel $B_x = 29\,868$, $B_y = 23\,835$ und $B_z = 12\,658$ die gestellten Anforderungen.

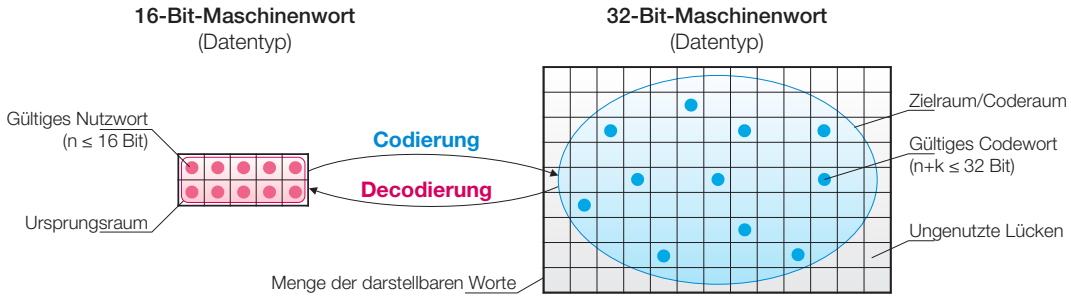


Abbildung 4.3: Binäre Darstellung nicht-systematischer arithmetischer Codes.
Der im Zuge der Codierung entstehende Coderaum (rechts) wird in seiner binären Form auf ein Maschinenwort abgebildet, in diesem Beispiel 32 Bit breit. Von diesen werden bis zu 16 Bit für die Nutzdaten (Ursprungsraum, links) benötigt, die verbleibenden Bits können für Prüfinformationen genutzt werden. Da sich Zweierpotenzen als ineffektive Wahl für die Codierungsschlüssel A erweisen, kann der zur Verfügung stehende Wertebereich des Maschinenwortes jedoch nie vollständig durch den Coderaum ausgenutzt werden. Die entstehenden Lücken stellen eine Gefahr für die Integrität des Codes dar.

4.2.4 Binäre Darstellung nicht-systematischer arithmetischer Codes

Die bereits angesprochene binäre Darstellung der nicht-systematischen ANBD-Codes hat nicht nur einen direkten Einfluss auf die Hamming-Distanz der entstehenden Codewörter, sie erweist sich darüber hinaus als überaus problematisch für die technische Umsetzung der arithmetischen Codierung. So verweist Schiffel [Sch11b] aufgrund der Darstellung explizit auf den Einsatz von *Fehlersimulation* (engl. *fault simulation*) (vgl. Kapitel 4.2.5) anstelle der Hamming-Distanz, um die Erkennungsleistung eines bestimmten Codierungsschlüssels zu ermitteln.

Abweichende Ergebnisse in der Literatur

Tatsächlich weichen die im vorangegangenen Abschnitt gefundenen Ergebnisse für die Hamming-Minimaldistanz teilweise von den Werten in der Literatur ab. Die von Schiffel [Sch11b] für eine geringe Zahl an Schlüsseln durchgeführten Experimente, weisen mitunter Restfehlerwahrscheinlichkeiten für Fehlermuster aus, die nach der Hamming-Distanz gleichsam als fehlerfrei erwartet werden. Diese Abweichungen stellen im Umkehrschluss die Gültigkeit und Anwendbarkeit der Hamming-Distanz in Frage und verhindern damit potenziell auch die Auswahl eines geeigneten As für die Erkennung einer bestimmten Zahl von Bitfehlern. Eine Wiederholung der von Schiffel durchgeführten Experimente mittels Fehlersimulation ergab die selben Abweichungen in der erwarteten Restfehlerwahrscheinlichkeit und führte zu unerkannten Datenfehlern, die gemäß der Hamming-Minimaldistanz nicht hätten auftreten dürfen.

Fallstrick: Binäre Darstellung

Die Ursachen hierfür lassen sich auf die binäre Darstellung der Codewörter und deren Abbildung auf Maschinenworte zurückführen. Abbildung 4.3 zeigt dies am Beispiel eines 32-Bit-Codes für 16-Bit-Datenwörter. In diesem Beispiel werden für die eigentlichen Nutzdaten $n \leq 16$ Bit sowie $k \leq 32 - n$ Bit für den Codierungsschlüssel reserviert. Um die Codierung nicht zu einer bitweisen Verschiebung abzuwerten, muss zudem $A < 2^k$ gewählt werden. Dies führt jedoch zu der in Abbildung 4.3 (rechts) gezeigten unvollständigen Ausnutzung des für die Darstellung des Codewortes genutzten 32-Bit-Maschinenwortes. Vereinfacht ausgedrückt generieren arithmetische Codes konzeptbedingt Coderäume, welche sich nicht exakt auf eine Zweierpotenz und damit auf eine feste Anzahl an Bits abbilden lassen. Die entstehenden Lücken sind ihrerseits nicht-systematisch und

```
4  function DECODE(v_c, A, B, D)
5      if v_c > v_{c,max} then SIGNAL_DUE()      ⇝ Coderaum: Codierungsschlüssel
6      if v_c mod A ≠ (B + D) then SIGNAL_DUE()
7      return (v_c − (B + D)) div A
8  end function

22 function APPLY(v_c, B_dyn)
23     if B_dyn > B_max then SIGNAL_DUE()        ⇝ Coderaum: Signaturen
24     return v_c + B_dyn
25 end function
```

Programmausschnitt 4.1: Schwachstelle binäre Darstellung – Integrität des Coderaumes.
CoRed erweitert die gebräuchliche Decodierungsfunktion (`decode()`, oben) von Forin [For89] um die Bereichsprüfung in Zeile 5. Diese einfache Überprüfung des codierten Wertes v_c auf die hierfür zulässige Obergrenze $v_{c,max}$ verhindert effektiv die Verletzung des durch den Codierungsschlüssel A generierten Coderaums durch Bitfehler in den Lücken der binären, nichtsystematischen Darstellung der Codewörter. In ähnlicher Weise dient die Überprüfung der Signatur B_{dyn} in Zeile 23 der CoRed-eigenen Funktion `apply()` der Integrität des durch die Sprungsignaturen des Mehrheitsentscheiders erwarteten Coderaums.

lassen sich weder bestimmten Bitpositionen noch einer gegebenen Anzahl an Bits zuordnen. Die Codierungstheorie berücksichtigt die binäre Abbildung der Codewörter jedoch nicht explizit, sondern erwartet einen in sich abgeschlossenen Coderaum und Wertebereich. In der Folge können auch Bitfehler unterhalb der durch die Hamming-Distanz berechneten Erkennungsschwelle zu unerkannten Datenfehlern führen, falls sie die ungenutzten Bits eines Codewortes betreffen. Diese Art der Mutation kann zu einem der Codierungsvorschrift entsprechenden jedoch *ungenutzten* Codewort mit $v > 2^n$ führen. Im Zuge der Decodierung entsteht hieraus ein schwer zu ermittelnder Überlauf beziehungsweise Unterlauf in dem für die Nutzdaten verwendeten Maschinenwort.

Zuverlässige Fehlererkennung bei binärer Darstellung

Die Problematik der unvollständigen Ausnutzung der Maschinenworte durch den durch die arithmetische Codierung gebildeten Coderaum wird in der Literatur (vgl. Arbeiten zu arithmetischer Codierung in Kapitel 2.2.2) bis jetzt nicht explizit adressiert. Die verfügbaren Daten zu den tatsächlichen Restfehlerwahrscheinlichkeiten legen vielmehr den Schluss nahe, dass die bestehenden Ansätze den allgemeinen Entwurfsvorgaben der Codierungstheorie folgen und die binäre Darstellung entsprechend nicht oder nur unzureichend berücksichtigen.

CoRed löst diesen Fallstrick hingegen durch die fallspezifische Begrenzung der Codewörter auf den tatsächlichen Coderaum. Dies führt letztlich zu der in Programmausschnitt 4.1 hervorgehobenen zusätzlichen Überprüfung der Coderaumgrenzen in Zeile 5 der Decodierungsfunktion `decode()` (oben). Die Obergrenze $v_{c,max}$ lässt sich hierbei vorab aus der Größe des Ursprungsraums, dem Codierungsschlüssel sowie der eingesetzten Signaturen berechnen. Darüber hinaus müssen zusätzlich auch die Sprungsignaturen des CoRed-Mehrheitsentscheiders entsprechend berücksichtigt werden, da diese, wie in Kapitel 4.2.3 beschrieben, durch CoRed als eigenen Code aufgefasst werden. Daher muss diese CoRed-spezifische Erweiterung des Mehrheitsentscheiders in gleicher Weise vor einem Überlauf beziehungsweise Unterlauf des *Coderaums der Signaturen* geschützt werden. Zu diesem Zweck führt CoRed die in Zeile 23 hervorgehobene Überprüfung in die `apply()`-Funktion (unten) des CoRed-Mehrheitsentscheiders ein.

Durch die Berücksichtigung der binären Darstellung der Codewörter im Entwurf der arithmetischen Codierung werden letztlich die Ergebnisse der Hamming-Distanz bestätigt, weshalb diese tatsächlich ein *valides Kriterium für die Auswahl des Codierungsschlüssels* darstellt. Darüber hinaus kann die einfache Überprüfung der Coderaumgrenzen im Allgemeinen eine signifikante

Herabsetzung der Zuverlässigkeit der Codierung verhindern. Die untere Schranke für die Restfehlerwahrscheinlichkeit (p_{sdc}) in einem 64-Bit-Code mit dem Codierungsschlüssel $A = 251$ beläuft sich beispielsweise auf näherungsweise 0,3 Prozent (vgl. Gleichung 4.1). Ohne die Beachtung der binären Darstellung ergibt sich hingegen der auch in der Literatur [Sch11b] zu findende Messwert von $p_{sdc} \approx 1{,}3$ Prozent. Dieser Wert sinkt durch die in CoRed realisierte Umsetzung der arithmetischen Codierung auf 0,0015 Prozent ab. Dies entspricht einer Verbesserung der Zuverlässigkeit um den Faktor 1000 gegenüber gängigen Implementierungen.

Binäre Darstellung der Codewörter als Schwachstelle

Durch arithmetische Codierung erzeugte Coderäume lassen sich nicht exakt auf Maschinenworte abbilden. Die entstehenden Lücken werden von der Codierungstheorie nicht erwartet. Im Gegensatz zu anderen Ansätzen berücksichtigt CoRed diese in der Implementierung und verhindert so einen signifikanten Verlust an Zuverlässigkeit sowie unvorhergesehene unerkannte Datenfehler. Darüber hinaus stellt dies die Gültigkeit der Hamming-Distanz sicher.

4.2.5 Verbesserung der Erkennungsleistung bei Überbeanspruchung

Die minimale Hamming-Distanz eines Codes erlaubt wie erwartet die Bestimmung der Anzahl der durch diesen zuverlässig erkennbaren Bitfehler. Steigt die Fehlerzahl jedoch über diese Schwelle hinaus, sind weiterhin unerkannte Datenfehler möglich. An dieser Stelle greift wiederum die Restfehlerwahrscheinlichkeit (p_{sdc}) als einfach zu bestimmendes und weithin genutztes Maß für die Zuverlässigkeitsabschätzung [Fro77].

Erst die *Fehlersimulation* ermöglicht die Bestimmung der Effektivität der Codierung bei einer Überschreitung der maximalen Fehlerzahl, wie sie im vorangegangen Abschnitt am Beispiel des Doppelbitfehlers aufgezeigt wurde. Die Simulation testet jedes mögliche Fehlerszenario und spannt hierfür entsprechend den *gesamten Fehlerraum* (engl. *fault space*) auf. Dieser setzt sich im Allgemeinen aus jeder möglichen Kombination von Codierungsschlüssel (A), Wert (v) und Bitfehlermuster zusammen. Die folgenden Experimente betrachten dabei einen 32-Bit-Code mit jeweils 16 Bit großen Codierungsschlüsseln und Werten. Um die Komplexität in handhabbaren Grenzen zu halten, wurden die Fehlermuster in diesem Fall auf maximal 8-fach-Bitfehler beschränkt. Der hierdurch entstehende Fehlerraum umfasst, trotz der Einschränkung, die beträchtliche Zahl von $6{,}5 \cdot 10^{16}$ Experimenten. Der hierfür notwendige Berechnungsaufwand entsprach einer Laufzeit von über 8,4 Jahren auf einem Einzelsystem[1]. Zum Vergleich: Die Berechnung der Hamming-Distanz für die selbe Zahl an möglichen Codierungsschlüsseln erforderte lediglich 70 Stunden. In der praktischen Umsetzung des Auswahlprozesses ist die Fehlersimulation daher vor allem als zusätzlicher Schritt nach einer Vorauswahl anhand der Hamming-Minimaldistanz zu sehen.

Einfluss des Codierungsschlüssels auf die Restfehlerwahrscheinlichkeit

Die Ergebnisse weisen im Allgemeinen das erwartete Verhalten auf und belegen insbesondere auch die absolute Übereinstimmung mit den in Kapitel 4.2.2 errechneten minimalen Hamming-Distanzen der jeweiligen Codes. Darüber hinaus zeigt sich jedoch abermals der signifikante Einfluss des Codierungsschlüssels auf die tatsächliche Restfehlerwahrscheinlichkeit sowie die bereits beobachtete Streuung der Ergebnisse. Abbildung 4.4 stellt dieses Verhalten anhand einer stark

[1] Im Bezug auf die Leistungsfähigkeit eines Intel Xeon 5650 Kerns mit 2,66 GHz.

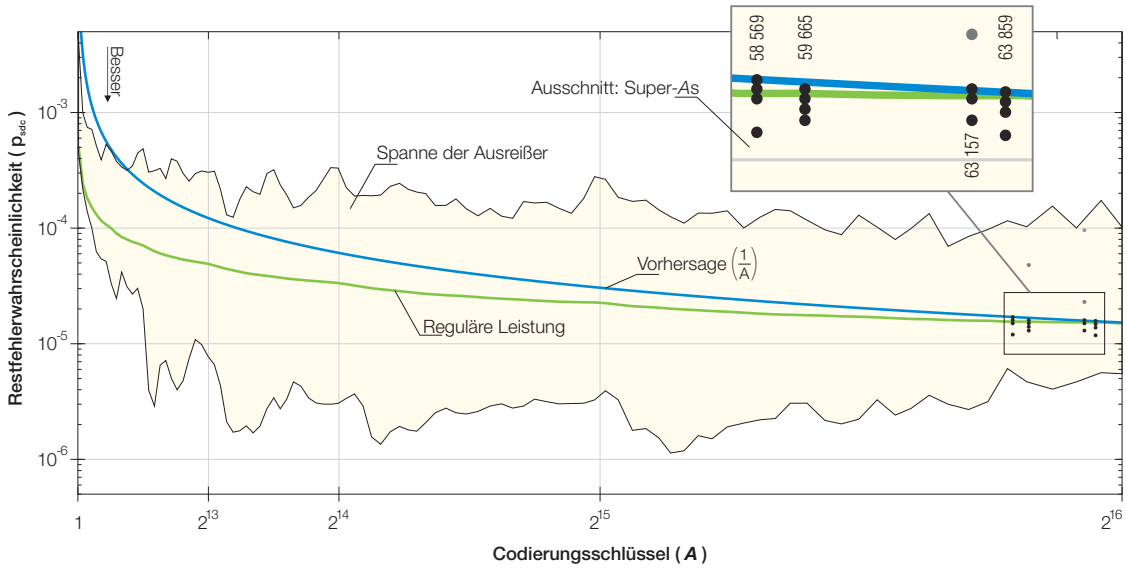


Abbildung 4.4: Vorhersage versus Messung – Streuung der Restfehlerwahrscheinlichkeit bei Überbeanspruchung. Vergleich der theoretischen *Vorhersage* der Restfehlerwahrscheinlichkeit mit den Messwerten der Fehlersimulation. Die *reguläre Leistung* entspricht einer deutlichen Überbeanspruchung der Codes (7 bis 8 Bit). Sie liegt unterhalb der Vorhersage und erweist sich als weitgehend frei von Streuung. Demgegenüber führt gerade die geringfügige Überbeanspruchung der Codierung zu *Ausreißern* an den Grenzen zur sicheren Fehlererkennung. Die *Spanne der Ausreißer* veranschaulicht die in diesem Bereich auftretende Streuung (Minimum/Maximum), welche die Vorhersage teilweise deutlich übersteigt. Der *Ausschnitt* zeigt zudem die Einordnung der individuellen Messwerte nach der Zahl der Bitfehler am Beispiel der Super-*A*s.

vereinfachten Darstellung der Messergebnisse für 16-Bit-Primzahlen als Codierungsschlüssel dar – ohne den Verlust der Allgemeinheit erlaubt die geringere Anzahl an Messpunkten eine deutlich intuitivere Visualisierung. Der Graph zeigt für jeden Schlüssel (A) die experimentell ermittelten Fehlerwahrscheinlichkeiten für 1-fach- bis 8-fach-Bitfehler. Die Vergrößerung illustriert die einzelnen Messpunkte am Beispiel der Super-*A*s aus Kapitel 4.2.2.

Grundsätzlich günstig ist die *reguläre Leistung* der Codes. Diese bezieht sich auf eine hohe Zahl von Bitfehlern im Bereich von sieben und acht Bits, welche oberhalb der Hamming-Minimaldistanz liegen und einer *deutlichen* Überbeanspruchung des jeweiligen Codes entsprechen. Unabhängig von dem gewählten Codierungsschlüssel liegt die Restfehlerwahrscheinlichkeit über den gesamten Wertebereich unterhalb der allgemeinen Vorhersage von p_{sdc} und ist darüber hinaus weitgehend frei von Streuungen und Schwankungen – die Messpunkte ergeben näherungsweise eine glatte Kurve. Aus dieser Beobachtung heraus verbessern größere Codierungsschlüssel die Erkennungsleistung wie erwartet.

Ein gegenteiliges Bild ergibt sich jedoch in den *Grenzfällen* (engl. *borderline bit errors*), welche den Übergang von der zuverlässigen Fehlererkennung hin zu einer *geringfügigen* Überbeanspruchung der Codierung darstellen. Da die tatsächliche Grenze von der Hamming-Distanz des jeweiligen Codes abhängt, fasst der Graph entsprechend die Bitfehler von zwei bis vier Bit zusammen und stellt den Bereich zwischen dem jeweils größten und kleinsten Messwert dar. Abhängig von dem gewählten Codierungsschlüssel variiert die Restfehlerwahrscheinlichkeit in diesem Bereich teilweise um ein Vielfaches und ergibt die dargestellte *Spanne der Ausreißer*. Wie schon bei der Bestimmung der Hamming-Distanz, erschweren diese Ausreißer eine einfache Auswahl geeigneter Codierungsschlüssel. Problematisch ist in diesem Zusammenhang vor allem die teilweise Überschreitung der vorhergesagten Restfehlerwahrscheinlichkeit als angenommene Schranke.

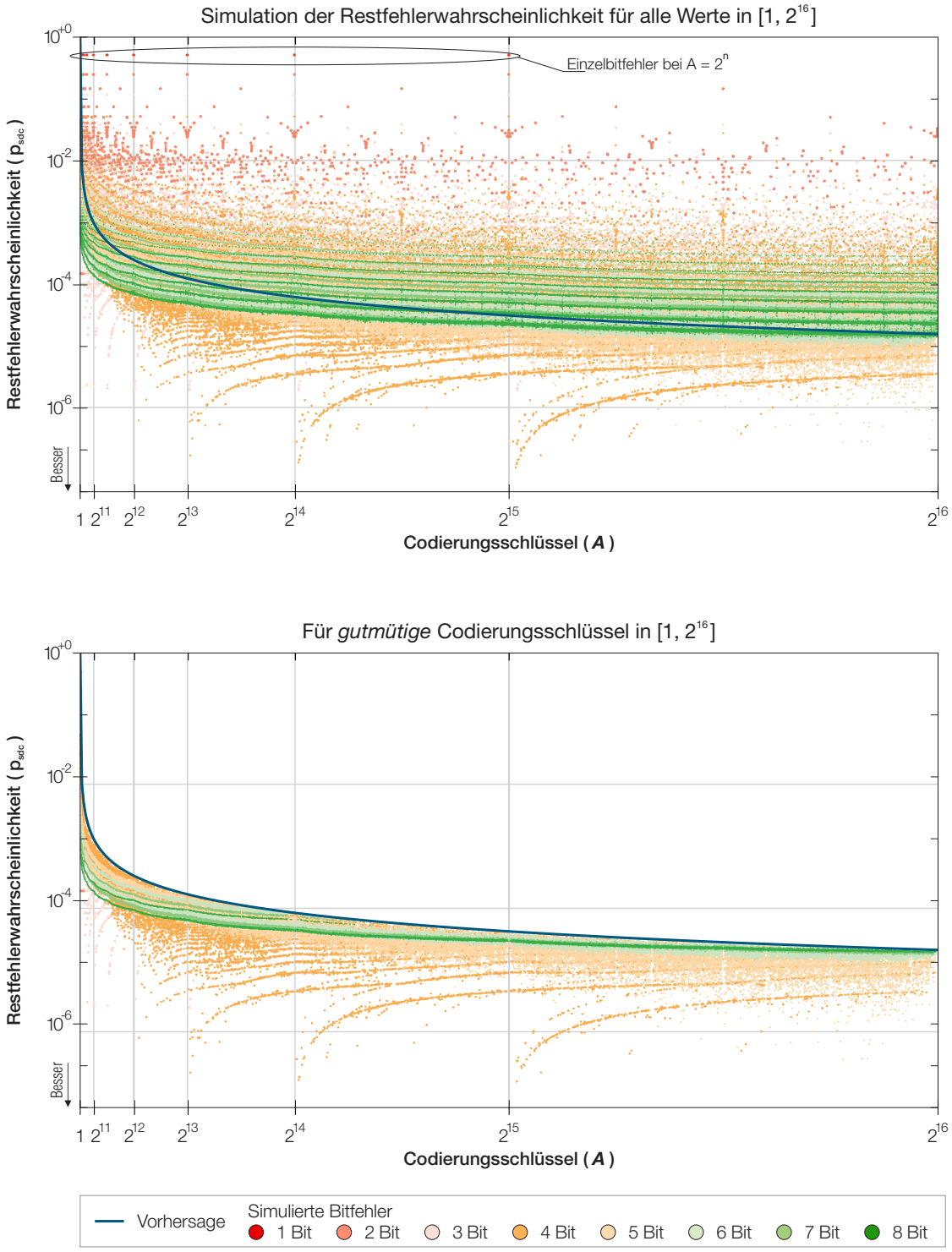


Abbildung 4.5: Vollständige Simulation für 32-Bit-Codes und bis zu 8-fach-Bitfehler.
Durch Fehlersimulation ermittelte Restfehlerwahrscheinlichkeit für alle 16-Bit-Codierungsschlüssel (oben). Die Messpunkte zeigen insbesondere für Fehlermuster von ein bis vier Bit eine signifikante Streuung, welche größtenteils die *Vorhersage* der Restfehlerwahrscheinlichkeit übersteigt. Wiederum zeigen sich die Unzulänglichkeiten der Zweierpotenzen durch die schlechteste Erkennungsrate aller Schlüssel. Die Menge der *gutmütigen A*s (unten) hält die Restfehlerwahrscheinlichkeit (p_{sdc}) als untere Schranke ein und weist eine deutlich geringe Streuung auf.

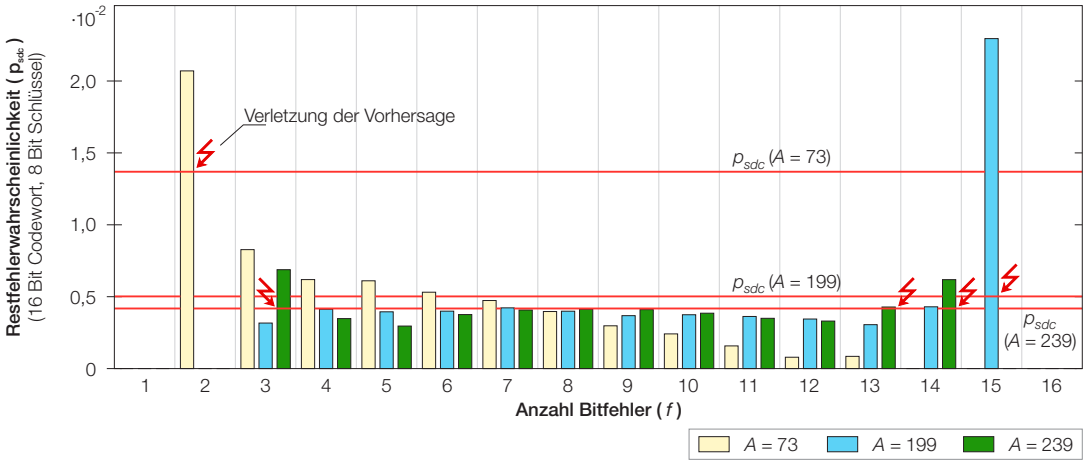


Abbildung 4.6: Verteilung der Restfehlerwahrscheinlichkeit in Abhängigkeit von der Bitfehleranzahl für 16-Bit-Codes. Eine detaillierte Betrachtung der tatsächlichen Restfehlerwahrscheinlichkeit in Abhängigkeit von der Anzahl der Bitfehler (1 bis 16 Bit). Die dargestellten Schlüssel (A) sind sämtlich *tückisch* und veranschaulichen drei typische Muster, die bei einer Überbeanspruchung des Codes entstehen: Verletzung der Vorgabe an der Vorderkante, Hinterkante oder beidseitig.

Über die vereinfachte Darstellung hinaus zeigt Abbildung 4.5 die Ergebnisse der Fehlersimulation als Ganzes. In dieser Darstellung überwiegt die enorme Streuung der Messpunkte, jedoch zeigen sich auch hier die erläuterten Muster für die reguläre Leistung. Die Unzulänglichkeiten der Zweierpotenzen, welche als einzige Schlüssel Einzelbitfehler zulassen, sind ebenfalls zu erkennen.

Eine detaillierte Betrachtung der auftretenden Ausreißer ermöglicht letztlich nur eine vollständige Fehlersimulation aller Fehlermuster. Durch den exponentiell wachsenden Aufwand für die Berechnung basiert die folgende Betrachtung auf 16-Bit-Codes und 8-Bit-Schlüssel. Abbildung 4.6 zeigt nunmehr die konkrete Verteilung der Restfehlerwahrscheinlichkeit in Abhängigkeit von der Anzahl der Bitfehler am Beispiel der drei dargestellten Codierungsschlüssel 73, 199 und 239. Diese stehen repräsentativ für die typischen Muster, welche bei einer Überbeanspruchung der arithmetischen Codierung entstehen. Zunächst ist die in allen drei Fällen vergleichsweise beständige Erkennungsleistung im mittleren Fehlerbereich gut zu erkennen. Dieses Verhalten erklärt die streuungsarme Kurve für diese Fehlerklasse in Abbildung 4.4. Im Vergleich dazu steigen die Werte an den Außengrenzen zur zuverlässigen Fehlererkennung an. Die für die Codewörter errechneten Vorhersagen für p_{sdc} sind daher als vertikale Referenzlinien eingezeichnet. Für $A = 73$ befinden sich der Ausreißer an der Vorderkante bei $f = 2$, während dies für $A = 199$ an der Hinterkante bei $f = 15$ der Fall ist. Ebenso können auch beide Seiten betroffen sein, wie für $A = 239$. Eine mögliche Erklärung für die mangelnde Effektivität einiger Codes in den Grenzfällen liegt in den Abständen der Codewörter. Entsprechen diese der Bitfehlerzahl, ergibt sich eine höhere Trefferquote für eine erfolgreiche Mutation der Codewörter.

Gutmütige und tückische Codes

In den in Abbildung 4.6 gezeigten Fällen verhalten sich die Codes durch die Ausreißer *tückisch* (engl. *malicious*), da sie die Vorhersage in bestimmten Situationen verletzen. Dies stellt zwar kein grundsätzliches Problem für die Codierung dar, erschwert jedoch die Zuverlässigkeitsabschätzung unnötig. Darüber hinaus ist die Aussicht bei einer Überbeanspruchung des Codes zunächst auf die Grenzfälle zu treffen gut, da die Wahrscheinlichkeit für Mehrfachbitfehler mit der Anzahl der Bits im Allgemeinen abnimmt [MHZA03, LSHC07, NDO11]. Entsprechend ist ein tückischer

Codierungsschlüssel trotz eines großen Hamming-Abstand unter Umständen weniger geeignet, da eine Überbeanspruchung zu einer inakzeptabel hohen Restfehlerwahrscheinlichkeit führt. Glücklicherweise verhält sich eine große Zahl von Codes insgesamt *gutmütig* (engl. *benign*) und verletzt die vorhergesagte Restfehlerwahrscheinlichkeit in keinem Fall. Im Falle der 16-Bit-Codes beläuft sich dieser Anteil auf 33 Prozent. Von den Super-*A*s der 32-Bit-Codes erweisen sich 58 659, 63 859 und 63 877 als gutmütig, lediglich 59 665 und 63 157 zeigen Ausreißer an der Hinterkante bei 27 und 28 Bit. Letztlich erfordert die Auswahl eines geeigneten Codierungsschlüssels auch die Berücksichtigung der Verteilung der Restfehlerwahrscheinlichkeiten. Ein sinnvoller Schritt ist zudem eine Beschränkung auf gutmütige Codierungsschlüssel, wie in Abbildung 4.5 (unten) gezeigt. Hierdurch fallen nicht nur die Streuungen deutlich geringer aus, es ermöglicht darüber hinaus die einfache Abschätzung der Restfehlerwahrscheinlichkeit durch p_{sdc} als untere Schranke.

Quintessenz: Auswahl der Codierungsparameter

Insgesamt haben die Codierungsparameter einen signifikanten Einfluss auf die Zuverlässigkeit der arithmetischen Codierung. CoRed nutzt hierfür einen zweistufigen Prozess: Vorauswahl durch die *Hamming-Minimaldistanz* und anschließende Optimierung anhand des Verhaltens (*gutmütig/tückisch*) bei einer Überbeanspruchung.

4.2.6 Resümee

Insgesamt haben die Codierungsparameter einen erheblichen Einfluss auf die Zuverlässigkeit der arithmetischen Codierung. Die Fallstricke und Besonderheiten in der binären Darstellung der Codierung sorgen in diesem Zusammenhang für eine erhebliche Streuung in der Erkennungsleistung und verhindert damit eine einfache Auswahl anhand der Schlüsselgröße. Eine Lösung bietet der in CoRed verfolgte zweistufige Prozess aus *Hamming-Distanz* und *Restfehlerwahrscheinlichkeit* (Gutmütigkeit) der Codierungsschlüssel. Entscheidend für dessen Anwendbarkeit ist die Berücksichtigung der Coderaumgrößen in der technischen Umsetzung der arithmetischen Codierung. CoRed erlaubt letztlich eine vollständige *und* zuverlässige Fehlererkennung in den Codewörtern bis zur minimalen Hamming-Distanz des jeweiligen Codes und verhindert darüber hinaus einen unnötigen Verlust der Erkennungsleistung.

4.3 Evaluation der CoRed Implementierung

Die Fehlersimulation beleuchtet bislang den Schritt vom theoretischen Entwurfskonzept zu einer technisch äquivalenten Umsetzung der arithmetischen Codierung. Transiente Fehler in Codewörtern werden durch CoRed im Zuge der Decodierung bis zu einer Bitfehlerzahl von $d_h - 1$ sicher erkannt. Diese Betrachtung beschränkt sich jedoch ausschließlich auf den Datenfluss von der Codierung bis zur Decodierung. Für einen analogen Funktionsnachweis des CoRed-Mehrheitsentscheiders ist jedoch zusätzlich dessen Kontrollfluss von entscheidender Bedeutung. Hierfür ausschlaggebend ist letztlich das durch den *Compiler* erzeugte *Binärprogramm* (engl. *binary*) sowie dessen Ausführung auf einer spezifischen Plattform.

Grundsätzlich weist der CoRed-Mehrheitsentscheider auch in Bezug auf mögliche Kontrollflussfehler eine konzeptionell vollständige Fehlererkennung auf. Deren Zuverlässigkeit ist im Folgenden Gegenstand einer eingehenden experimentellen Untersuchung durch Techniken der *Fehlerinjektion* (engl. *fault injection*) des Hardwarekontextes. Auch hier ist die Abdeckung des kompletten Fehlerraums das entscheidende Kriterium für den Nachweis einer zuverlässigen Fehlererkennung.

Durch die Fehlersimulation wurde bereits der aus $6{,}5 \cdot 10^{16}$ Möglichkeiten bestehende Fehlerraum aus Codierungsschlüsseln (A), Werten (v) und Fehlermustern abgedeckt. Der im Zuge der Fehlerinjektion entstehende Fehlerraum ergibt sich nunmehr aus der zeitlichen Abfolge der Instruktionen sowie der möglichen Fehlerstellen in Form von Registern und Speicherzellen.

Der folgende Abschnitt führt hierfür zunächst in den Versuchsaufbau und die elementaren Konzepte der Fehlerinjektion ein. Darauf aufbauend erfolgt die grundlegende Evaluation des CoRed-Mehrheitsentscheiders durch eine vollständige Betrachtung aller möglichen Einzelbitfehler. Die Ergebnisse veranschaulichen darüber hinaus den Einfluss stiller Annahmen der Ausführungsumgebung auf die Zuverlässigkeit des CoRed-Mehrheitsentscheiders. Die Erweiterung des Fehlerraums auf die Klasse der Mehrfachbitfehler ermöglicht schließlich die erneute Überprüfung der durch die Fehlersimulation erwarteten Fehlererkennungsleistung.

4.3.1 Fehlerinjektion auf der Befehlssatzebene

Die Injektion von Fehlern auf der Befehlssatzebene ermöglicht die gezielte Verfälschung von Instruktionen und Speicherzellen als den kleinsten Strukturelementen der Software [KKA95, BP03]. Die Fehlerinjektion auf dieser Ebene erlaubt folglich die direkte Überprüfung des in Kapitel 2.1.1 eingeführten Fehlermodells durch die Erzeugung fehlerhafter Operanden, Operatoren und Berechnungsergebnissen. Jedes Einzelexperiment setzt sich hierfür zusammen aus *Fehlermuster*, *Fehlerort* (Register oder Instruktion) und *Zeitpunkt* [AAA+90].

Das FAIL* Fehlerinjektionswerkzeug

Für die Evaluation auf der Befehlssatzebene wurde das FAIL* *Fehlerinjektionswerkzeug* [SHK+12] genutzt. Dieses basiert auf dem *Bochs Emulator* [Law96] und erlaubt die strukturierte und reproduzierbare Injektion von Fehlern in den Hardwarekontext im Zuge der Ausführung der Software. Die Beschreibung einer *Fehlerinjektionskampagne* (engl. *fault-injection campagain*) ermöglicht dabei die automatisierte Generierung von Einzelexperimenten zur systematischen Abdeckung des gesamten Fehlerraums. Die technische Umsetzung von FAIL* untergliedert sich in einen Server für die Verwaltung der Kampagne (engl. *campaign manager*) sowie einer Menge an Clients, welche unabhängig voneinander die Einzelexperimente durchführen. Durch die mögliche Verteilung der Aufgaben auf mehrere Rechner sind somit auch umfangreiche Kampagnen sinnvoll durchführbar.

Reduzierung des Fehlerraums

Der sich aus der Fehlerinjektion ergebende Fehlerraum ist üblicherweise sehr groß und bewegt sich selbst für die vergleichsweise geringe Größe des CoRed-Mehrheitsentscheiders und die angenommenen Einzelbitfehler bereits im Bereich um 10^5. Ein Großteil der möglichen Fehler manifestiert sich aus Sicht der Software jedoch als gutmütige Defekte (vgl. Kapitel 2.1.1), welche sich im Rahmen der Ausführung nicht auswirken und verdeckt werden. Es lassen sich hierbei *idempotente* und *unwirksame* Fehler unterscheiden. Letztere betreffen Fehlerorte, die im weiteren Verlauf nicht genutzt oder im Zuge der Ausführung überschrieben werden. Die Injektion eines Registers zwischen einer lesenden und einer schreibenden Instruktion ist ein Beispiel hierfür. Demgegenüber haben zueinander idempotente Fehler zwar einen Effekt auf die Ausführung, sie resultieren jedoch alle in der selben Fehlerauswirkung. Zum Beispiel spielt der Zeitpunkt der Fehlerinjektion zwischen einer schreibenden und einer lesenden Instruktion keine Rolle. Es genügt ein Fehler der entsprechenden Äquivalenzklasse, um die gewünschte Auswirkung zu erreichen.

Ohne den Verlust der vollständigen Abdeckung des Fehlerraums erlaubt FAIL* die Eliminierung von unwirksamen beziehungsweise idempotenten Fehlern und damit eine *Reduzierung des Fehler-*

raums (engl. *fault-space pruning*) auf die Klasse der *effektiven Fehler* (engl. *effective faults*). Hierfür extrahiert und analysiert FAIL* die Instruktionen des Binärprogramms, um daraus die tatsächliche Nutzung der Register und Speicherstellen zu ermitteln. Die notwendigen Experimente ergeben sich anschließend schrittweise aus einem *fehlerfreien Testlauf* (engl. *golden run*). Hierbei definiert jede Instruktion des Kontrollflusspfads einen Injektionszeitpunkt und jede Speicherstelle einen entsprechenden Ort. Die als effektiv ermittelten Experimente werden in eine Datenbank für die spätere Verarbeitung durch die FAIL*-Clients abgelegt.

4.3.2 Implementierung und Versuchsaufbau

Zur Durchführung der Experimente muss der CORED-Mehrheitsentscheider bereits als Binärcode vorliegen. Der von FAIL* eingesetzte Bochs Emulator gibt dabei die Hardwareplattform in Gestalt der *Intel 32-Bit-Architektur* (engl. *Intel Architecture 32-Bit*) (IA32) vor. Hinzu kommt eine geeignete Kampagnenbeschreibung, welche die möglichen Ergebnisse der Fehlerinjektion definiert und darüber hinaus die Ausführung der Experimente koordiniert.

Realisierung des CORED-Mehrheitsentscheiders

Die für die Evaluation genutzte Implementierung des CORED-Mehrheitsentscheiders sowie der arithmetischen Codierung liegt als C++-Quellcode vor. Sie nutzt die durch die Programmiersprache bereitgestellten Template-Sprachelemente (dt. *Schablonen*) für die Auswahl, Berechnung und Konfiguration der konstanten Codierungsparameter. Die hierfür notwendige Verwaltung der Schlüssel und Signaturen erfolgt durch eine in den *Erstellungsprozess* (engl. *build process*) integrierte Werkzeugkette zur Generierung einer nach den in Kapitel 4.2 vorgestellten Auswahlprinzipien besetzten *Nachschlagetabelle* (engl. *look-up table*). Darüber hinaus erfolgt die Belegung der Zeitstempel im Sinne eines Rundenanzeigers, welcher es dem Verarbeitungsstrom erlaubt zwischen der aktuellen und der vorangegangenen Periode zu unterscheiden.

Mehrheitsentscheider:	**Einfach**	**CoRed**
Instruktionen	38	92
Binärcode (Größe)	112 Byte	301 Byte

Tabelle 4.2: IA32-Plattform: Größenvergleich einfacher versus CORED-Mehrheitsentscheider (32-Bit-Codewörter).

Für die Evaluation wurden sowohl der CORED-Mehrheitsentscheider als auch sein ungeschütztes Pendant für die IA32 übersetzt. Hierfür kam der C++-Compiler der *GNU-Compilersammlung* (engl. *GNU Compiler Collection*) (GCC) in der Version 4.7.2-5 (Debian Linux) zum Einsatz. Die Compileroptionen entsprachen dabei mit Ausnahme der Optimierungsstufe (`-O2`, volle Optimierung) den Standardeinstellungen. Die sich ergebenden Größenverhältnisse der jeweiligen Binärprogramme sind in Tabelle 4.2 dargestellt. Sowohl hinsichtlich der Größe als auch der Zahl der Instruktionen weist der CORED-Mehrheitsentscheider näherungsweise die zweieinhalbfache Größe des einfachen Entscheiders auf. Die für die Bestimmung des Fehlerraums relevanten Registeroperationen setzen sich hierbei zusammen aus 14 16-Bit- und 240 32-Bit-Operationen.

Versuchsaufbau

Um eine vollständige *Zweigüberdeckung* (engl. *branch coverage*) bei der Fehlerinjektion zu erreichen, wurden die Mehrheitsentscheider in jeder Kampagne gegen die fünf möglichen Kombinationen der Konsensmenge ($x = y = z$; $x \neq y = z$; $y \neq x = z$; $z \neq x = y$; $x \neq y \neq z$) getestet. Da die

Effektivität der Fehlererkennung für verschiedene Eingabewerte (v) und Schlüssel (A) bereits durch die Fehlersimulation in Kapitel 4.2.5 nachgewiesen wurde, beschränkte sich die Fehlerinjektion auf eine feste Menge an möglichen Eingabewerten.

Um die Fehlerinjektion zu vereinfachen, wurde der zur Verfügung stehende IA32 Befehlssatz auf die Teilmenge der Instruktionen beschränkt, welche direkt auf Registern arbeiten. Hierdurch werden alle Speicheroperationen ausschließlich über Register durchgeführt und sind entsprechend für die Fehlerinjektion sichtbar. Diese Eigenschaft ermöglicht beziehungsweise erleichtert die Reduktion des Fehlerraums durch FAIL*.

Zuletzt wurde die geforderte Isolation des Mehrheitsentscheiders hergestellt. Hierzu zählt sowohl eine feingranulare räumliche Abtrennung des durch den Mehrheitsentscheider genutzten Adressraums mithilfe des Betriebssystems als auch die Überwachung der maximalen Laufzeit durch einen in den Experimentablauf integrierten Funktionswächter von FAIL*.

Ergebnisse der Fehlerinjektion

Im Anschluss an die eigentliche Durchführung eines Experiments wird dessen Ausgang durch die Clients wiederum in der Datenbank abgelegt. Die Bewertung basiert schließlich auf einem Vergleich mit dem Ergebnis des initial durchgeführten fehlerfreien Testlaufs. Grundsätzlich klassifizieren sich die Auswirkungen der Fehlerinjektion gemäß der bekannten Fehlerfolgen:

Ohne Auswirkung: Der durch den injizierten Fehler verursachte Defekt wird im weiteren Verlauf verdeckt. Diese Auswirkung deutet entsprechend auf Redundanzen in der Anwendung hin.

Erkannte Fehler: Diese Klasse fasst alle Arten der Fehlersignalisierung zusammen, welche eine, gegebenenfalls auch spätere, Behandlung der Fehlersituation erlauben. Entsprechend untergliedern sich die erkannten Fehler in:

- **CoRed (arithmetische Codierung):** Der Fehler wurde durch die arithmetische Codierung bei der nächsten Verwendung beziehungsweise Überprüfung des Wertes erkannt.
- **Hardwareausnahme:** Die Hardware signalisiert ihrerseits einen unzulässigen Vorgang oder Fehler, wie zum Beispiel eine Division durch null (engl. *division by zero*) oder eine unzulässige Instruktion (engl. *illegal instruction*).
- **Schutzverletzung (Programmcode/Speicher):** Der injizierte Fehler führt zu einer durch die Hardware beziehungsweise das Betriebssystem erkennbaren Verletzung (engl. *access violation*) der räumlichen Isolation des Prozesses. Dies umfasst Sprünge aus dem lokalen *Programmcode* (engl. *out of scope jumps*) oder unzulässige *Speicherzugriffe* (engl. *illegal memory access*).
- **Terminüberschreitung:** Durch den injizierten Fehler kommt es zur Überschreitung des vorgegebenen Termins und damit zu einer Verletzung der temporalen Isolation. Diese wird wiederum durch die Hardware respektive das Betriebssystem signalisiert.

Unerkannte Datenfehler: Der injizierte Fehler wurde nicht erkannt und die Ausführung produziert fehlerhafte Ergebnisse.

Schutzverletzungen und Terminüberschreitungen werden zusammen mit sonstigen Hardwareausnahmen als erkannte, nicht behebbare Fehler klassifiziert, da sie sich grundsätzlich aktiv behandeln lassen – zum Beispiel durch die in Kapitel 3.6 vorgeschlagene Wiederholung des Mehrheitsentscheids.

```
1 function CoRed_vote(x_c, y_c, z_c)
2     zero_local_storage()
3     if (x_c − y_c) = (B_x − B_y) then
4         if (x_c − z_c) = (B_x − B_z) then
  ...
```

Programmausschnitt 4.2: Schwachstelle Laufzeitumgebung – Integrität der isolierten Ausführung.
Stille Annahmen der Laufzeitumgebung an die Binärschnittstelle können *im Fehlerfall* zu einer Verletzung der angestrebten Isolation führen, zum Beispiel durch einen Verbleib der Inhalte flüchtiger Register bei einem Kontextwechsel. Im Bedarfsfall sorgt die Funktion `zero_local_storage()` (Zeile 2) durch die Bereinigung des *gesamten* Hardwarekontextes vor der eigentlichen Ausführung des Mehrheitsentscheids für eine definierte Ausführungsumgebung.

4.3.3 Isolation der Verarbeitungsströme

Eine essentielle Forderung der Fehlerhypothese (vgl. Kapitel 3.2.1) ist die strikte Isolation der einzelnen CoRed-Komponenten beziehungsweise Einzelfehlerbereiche untereinander. Der Entwurf des CoRed-Ansatzes geht entsprechend davon aus, dass sich transiente Fehler nur innerhalb des eigenen Ausführungskontextes ausbreiten können. Demgegenüber werden Seiteneffekte durch andere Verarbeitungsströme nicht betrachtet. Die technische Umsetzung dieser Anforderungen erweist sich als teilweise schwierig. Eine lückenhafte Isolation kann in bestimmten Grenzfällen jedoch zu einem unerwarteten Versagen der Fehlertoleranz und damit zu unerkannten Datenfehlern führen. Im Zuge der Fehlerinjektion des Programmzählers (vgl. Kapitel 4.3.4) konnten insgesamt drei entsprechende Grenzfälle beobachtet werden.

Fallstrick: Stille Annahmen und Optimierungen der Ausführungsumgebung

Die Ursachen lassen sich auf stille Annahmen der Ausführungsumgebung hinsichtlich der *Binärschnittstelle* (engl. *application binary interface*) (*ABI*) der zugrunde liegenden Hardwareplattform zurückführen. Sowohl der eingesetzte Compiler, als auch das Betriebssystem können diese Spezifikation für valide Optimierungen nutzen, zum Beispiel um bei einem *Kontextwechsel* (engl. *context switch*) beziehungsweise Funktionsaufruf lediglich die Menge der *nicht-flüchtigen Register* (engl. *non-volatile registers*) des Hardwarekontextes zu sichern. Per Konvention sind die Inhalte der anderen Register ungültig und müssen vor der Verwendung zunächst initialisiert werden.

Bei der Übersetzung des CoRed-Mehrheitsentscheiders trifft der Compiler die selben stillen Annahmen und plant die Registernutzung entsprechend. Im Fehlerfall kann es jedoch durch eine Verfälschung des Kontrollflusses innerhalb des CoRed-Mehrheitsentscheiders zu einem Überspringen der konventionsgemäß notwendigen schreibenden Operationen vor der Nutzung der entsprechenden Werte kommen. In der Folge können sich *gültige Codewörter der Vorgängerkomponente* in den flüchtigen Registern befinden. Diese Verletzung der geforderten räumlichen Isolation kann in Kombination mit einem ungünstigen Kontrollflussfehler zu einer ordnungsgemäßen Weiterverarbeitung dieser *untoten Werte* (engl. *defunct value*) in den codierten Operationen des Mehrheitsentscheiders führen.

Zuverlässige Isolation

Die genannten stillen Annahmen und darauf aufbauende Optimierungen liegen zusammen mit der effektiven Isolation des Prozesskontextes grundsätzlich in der Verantwortung des Betriebssystems. Der Entwurf des CoRed-Mehrheitsentscheiders geht im Allgemeinen von einer *sauberen* Ausführungsumgebung aus. Im Einzelfall lässt sich das Verhalten des Betriebssystems beziehungsweise die korrespondierende Generierung des Binärprogramms durch den Compiler nicht direkt

		Register und Merker		Instruktionen		Programmzähler	
		Einfach	CoRed	Einfach	CoRed	Einfach	CoRed
Ohne Auswirkung (verdeckter Defekt)		1040	3204	784	2772	127	267
Erkannte Fehler	CoRed (arithmetische Codierung)	–	1435	–	995	–	420
	Hardwareausnahme	8	41	93	246	21	241
	Schutzverletzung (Programmcode)	173	559	149	208	2614	5614
	Schutzverletzung (Speicher)	1652	3177	676	1626	190	626
	Terminüberschreitung	0	0	0	1	0	0
Unerkannter Datenfehler		**807** ↯	0	**450** ↯	0	**152** ↯	0
	$\sum$	3680	8416	2152	5848	3104	7168

Tabelle 4.3: Fehlerinjektion von Einzelbitfehlern – Evaluation des CoRed-Mehrheitsentscheiders auf Befehlssatzebene. Injektion von Fehlern in den Hardwarekontext des CoRed-Mehrheitsentscheiders ($A = 58\,659$) untergliedert in die drei Bereiche *Register und Merker*, *Instruktionen* und *Programmzähler*. Die hier durchgeführte Kampagne deckt dabei den gesamten Fehlerraum für Einzelbitfehler ab. Die Ergebnisse zeigen dabei sowohl die Anfälligkeit des einfachen Mehrheitsentscheiders als auch die Robustheit des CoRed-Mehrheitsentscheiders gegenüber unerkannten Datenfehlern.

beeinflussen. In diesen Fällen ist eine plattformspezifische Erweiterung des CoRed-Mehrheitsentscheiders zweckmäßig, um einen Bruch der Isolation *im Fehlerfall* zu vermeiden. Im konkreten Beispiel erweitert CoRed den Mehrheitsentscheider um eine eigenständige Bereinigung des Hardwarekontextes durch die Funktion `zero_local_storage()` in Zeile 2 von Programmausschnitt 4.2.

Stille Annahmen der Ausführungsumgebung als Schwachstelle

Üblicherweise valide Optimierungen des Betriebssystems beziehungsweise des Compilers beim Wechsel des Hardwarekontextes ermöglichen eine Verletzung der Isolation *im Fehlerfall*. Der CoRed-Mehrheitsentscheider geht von einem *sauberen* Kontext aus. Im Bedarfsfall muss dieser durch eine plattformspezifische Umsetzung oder Anpassung hergestellt werden.

4.3.4 Prüfstein Einzelbitfehler

Für den entscheidenden Lackmustest und den Nachweis der Effektivität des CoRed-Ansatzes wurde sowohl der einfache Mehrheitsentscheider als auch der CoRed-Mehrheitsentscheider der im Folgenden näher beschriebenen Fehlerinjektion unterzogen. Die Experimente umfassten dabei den gesamten Fehlerraum für Einzelbitfehler. Generell ist zu bemerken, dass die Anzahl der Experimente aufgrund der Größenunterschiede in Programmcode und Speicher der beiden Varianten voneinander abweichen. Die Messergebnisse der Kampagne sind in Tabelle 4.3 dargestellt.

Überblick

Auffällig ist zunächst die große Zahl verdeckter Fehler, welche sich aus der bereits in Kapitel 3.5 angesprochenen inhärenten Redundanz eines Mehrheitsentscheiders ergibt. Nichtsdestotrotz zeigt sich die Anfälligkeit des einfachen Mehrheitsentscheiders in Bezug auf unerkannte Datenfehler, deren Anteil sich im Mittel auf über 20 Prozent der effektiven Fehler beläuft. Demgegenüber weist

der CoRed-Mehrheitsentscheider *keine unerkannten Datenfehler* mehr auf. Diese werden stattdessen zuverlässig durch die arithmetische Codierung aufgedeckt, wobei sich der mittlere Anteil aufgrund der unterschiedlichen Implementierungen auf nunmehr 19 Prozent der effektiven Fehler summiert. Für eine differenzierte Betrachtung gliedert sich die Fehlerinjektion in die folgenden drei Kategorien: *Register und Merker* (engl. *registers and flags*), *Instruktionen* (engl. *instructions*) und *Programmzähler* (engl. *program counter*).

Register und Merker

Die Fehlerinjektionsexperimente für die *allgemeinen Register* (engl. *general purpose register*) sowie die *Merker* (engl. *flags*) des Statusregisters stellen den zahlenmäßig größten Block der Einzelfehlerkampagne dar. Die überwiegende Zahl der effektiven Fehler mündet hierbei in eine Verletzung der räumlichen Isolation durch die auftretenden Verfälschungen der in den Registern abgelegten Adressen. Daneben kommt es lediglich vereinzelt zu Hardwareausnahmen durch Rechenfehler oder ungültige Werte des Statusregisters. Letztere weisen durch den in den bedingten Verzweigungen häufig genutzten *Nullmerker* (engl. *zero flag*) eine schwerwiegende Fehlerquelle auf, da dieser die Sprungentscheidung direkt beeinflusst. Der einfache Mehrheitsentscheider weist im Falle der Register und Merker über 30 Prozent unerkannte Datenfehler auf.

Instruktionen

Die Verfälschung von Instruktionen führt zu einem deutlich differenzierteren Fehlerbild. Die zu erwartenden Ausnahmefehler der Hardware für unzulässige Instruktionen bewegen sich lediglich im Bereich zwischen sieben und acht Prozent. Der Grund hierfür ist die aus Effizienzgründen üblicherweise knapp bemessene Wortbreite der Instruktionen, weshalb die meisten Werte tatsächlich gültige Anweisungen darstellen. Mutationen führen daher häufig zu Berechnungsfehlern und einer Fortsetzung der Ausführung. Entsprechend hoch ist die Anfälligkeit des einfachen Mehrheitsentscheiders mit annähernd 33 Prozent unerkannten Datenfehlern in dieser Kategorie. Daneben zeigen sich auch in hohem Maße Schutzverletzungen, zum Beispiel durch eine Veränderung der Adressierungsart. Aufgrund der Abwesenheit von Schleifen oder ähnlichen Kontrollstrukturen, sind Verletzungen der zeitlichen Isolation hingegen praktisch ausgeschlossen. Lediglich in einem pathologischen Fall kommt es zu einer Terminüberschreitung bei der Ausführung des CoRed-Mehrheitsentscheiders durch eine ungünstig verfälschte Sprungadresse.

> **Effektivität des CoRed-Mehrheitsentscheiders**
>
> Die technische Umsetzung des CoRed-Mehrheitsentscheiders benötigt im Vergleich zu einer Standardimplementierung lediglich 54 zusätzliche Instruktionen. Die Fehlerinjektion auf der Befehlssatzebene weist darüber hinaus eine *zuverlässige* und *vollständige* Fehlererkennung aller Einzelbitfehler nach. Der CoRed-Mehrheitsentscheider verhält sich insgesamt wie erwartet und *eliminiert die unerkannten Datenfehler gänzlich*.

Programmzähler

Die Injektion von Einzelbitfehlern in den Programmzähler führt letztlich zu zufälligen Sprüngen um eine Zweierpotenz als Adressabstand (engl. *offset*). Durch die relativ kleine Größe des Mehrheitsentscheiders führen diese Sprünge in der Regel aus dem aktuellen Kontext heraus und werden folglich durch die räumliche Isolation entlarvt. Dementsprechend ist die Fehlerrate für

		Doppelbitfehler in allgemeinen Prozessorregistern	
		Super $A = 58\,659$	Unsicheres $A = 58\,368$
Ohne Auswirkung (verdeckter Defekt)		38 639	38 639
Erkannte Fehler	CoRed (arithmetische Codierung)	21 596	21 519
	Hardware Ausnahme	47	47
	Schutzverletzung (Programmcode)	471	471
	Schutzverletzung (Speicher)	59 967	59 967
	Terminüberschreitung	0	0
Unerkannter Datenfehler		0	**77** ↯
	$\sum$	120 720	120 720

Tabelle 4.4: Überprüfung der Hamming-Distanz auf der Befehlssatzebene durch die Injektion von Doppelbitfehlern. Die Erweiterung des Fehlerraums auf Doppelbitfehler zeigt die Auswirkungen eines ungünstig gewählten Codierungsschlüssels. In diesem Beispiel weist das *unsichere* $A = 58\,368$ lediglich eine Hamming-Distanz von zwei auf. Wie erwartet treten in der Folge unerkannte Datenfehler auf. Demgegenüber erweist sich das Super-$A = 58\,659$ weiterhin als absolut zuverlässig.

unerkannte Datenfehler mit fünf Prozent für den einfachen Mehrheitsentscheider vergleichsweise gering. Dennoch zählen möglich Sprünge innerhalb des aktuellen Ausführungskontextes zu den schwerwiegenden Auswirkungen transienter Fehler, welche sich erst durch die Sprungsignaturen des CoRed-Mehrheitsentscheiders aufdecken lassen.

4.3.5 Erweiterte Betrachtung von Mehrfachbitfehlern

Die grundlegende Evaluation des CoRed-Mehrheitsentscheiders mittels Einzelbitfehlern zeigt die Übereinstimmung zwischen dem Entwurf und der technischen Realisierung. Darüber hinaus lässt sich die arithmetische Codierung jedoch nicht an ihre Leistungsgrenze bringen, da alle sinnvollen Codierungsschlüssel (ausser 2^k) eine minimale Hamming-Distanz von mindestens zwei aufweisen – Einzelbitfehler werden daher in jedem Fall erkannt. Erst die erweiterte Betrachtung von Mehrfachbitfehlern ermöglicht eine Analyse der Codierungsparameter auf der Befehlssatzebene. Die Leistungsfähigkeit des FAIL* Werkzeugs erlaubte entsprechende Experimente trotz des rasant wachsenden Fehlerraums.

Überprüfung der Hamming-Distanz auf der Befehlssatzebene

Da eine ganze Reihe von Codierungsschlüsseln eine geringe Hamming-Minimaldistanz von zwei aufweisen, erlauben bereits Doppelbitfehler die Überprüfung der Auswirkungen eines schlecht gewählten *A*s. In einem ersten Schritt wurde die Fehlerinjektion daher auf die Klasse der *Doppelbitfehler* erweitert. Der sich in diesem Fall ergebende Fehlerraum kann durch FAIL* auf eine handhabbare Zahl von 120 720 effektiven Fehlern reduziert werden. Daher ist die vollständige Abdeckung des Fehlerraums abermals gegeben.

Die Ergebnisse der Doppelbitfehlerkampagne sind in Tabelle 4.4 dargestellt. Wie zu erwarten treten bei einem geeignet gewählten $A = 58\,659$ keine unerkannten Datenfehler auf. Im Gegensatz dazu erweist sich die arithmetische Codierung bei einem schlecht gewählten $A = 58\,368$ mit einer minimalen Hamming-Distanz von $d_h = 2$ mit 77 unerkannten Datenfehlern als anfällig. Dass es sich hierbei um den erwarteten Einfluss des Codierungsschlüssels handelt, zeigt sich an der wertgleichen Reduzierung der durch die arithmetische Codierung erkannten Fehler.

Codierungsschlüssel (A)	**22**	**44**	**76**	**2404**	**12288**	**34346**	**58368**
d_h	2	2	2	2	2	2	2
p_{sdc} (Simulation, 2 Bit)	0,02749	0,02749	0,01300	0,00029	0,14247	0,00009	0,01633
Unerkannte Datenfehler	138	143	63	4	591	4	77

Tabelle 4.5: Überprüfung der Restfehlerwahrscheinlichkeit auf Befehlssatzebene durch die Injektion von Doppelbitfehlern. Der Vergleich der simulierten Restfehlerwahrscheinlichkeit (p_{sdc}) für Doppelbitfehler mit den Ergebnissen der entsprechenden Fehlerinjektionskampagnen zeigt den Einfluss des gewählten Codierungsschlüssels (A) bei einer Überbeanspruchung. Die Korrelation zwischen unerkannten Datenfehlern und der Restfehlerwahrscheinlichkeit beträgt 99,86 Prozent.

Überprüfung der Restfehlerwahrscheinlichkeit auf der Befehlssatzebene

Durch eine Ausweitung der Doppelbitfehlerkampagne auf weitere Codierungsschlüssel wurde zudem der Zusammenhang zwischen dem gewählten A und der resultierenden Restfehlerwahrscheinlichkeit (p_{sdc}) auf der Befehlssatzebene überprüft. Die Ergebnisse dieser Untersuchung sind in Tabelle 4.5 dargestellt. In dieser Auswahl weisen alle Schlüssel einen minimalen Hamming-Abstand von zwei auf. Entsprechend überstrapazieren die injizierten Doppelbitfehler die entstehenden Codes. Diese zeigen sich, wie erwartet, anfällig für unerkannte Datenfehler, wobei ihre Zahl erheblich schwankt. Die Korrelation zwischen der simulierten Restfehlerwahrscheinlichkeit und der gemessenen Zahl unerkannter Datenfehler ist mit 99,86 Prozent sehr hoch. Die Zahlen stützen somit die in Kapitel 4.2.5 vorhergesagte Streuung der Restfehlerwahrscheinlichkeit in Abhängigkeit des gewählten As auch für die konkrete Umsetzung des CoRed-Mehrheitsentscheiders.

Ausblick auf schwerwiegendere Mehrfachbitfehler

Durch die geeignete Wahl der Codierungsschlüssel lassen sich mit CoRed prinzipiell bis zu 5-fach-Bitfehler in 32-Bit-Codes zuverlässig erkennen. Ein entsprechender Nachweis ist jedoch aufgrund der mit der Bitfehlerzahl dramatisch zunehmenden Größe des Fehlerraums kaum mehr möglich. Dieser vergrößert sich von $3{,}59 \cdot 10^6$ für 3 Bit auf $2{,}90 \cdot 10^9$ im Falle der 5-fach-Bitfehler. Um dennoch einen gewissen Einblick in die Domäne der Mehrfachbitfehler zu gewinnen, wurden die Experimente für 3-fach bis 5-fach-Bitfehler mit einer zufälligen Generierung der Fehlermuster durchgeführt. Hierbei wurden die Fehlerorte und Zeitpunkte vollständig beibehalten und lediglich die Anzahl der generierten Fehlermuster pro Injektionspunkt reduziert. Tabelle 4.6 zeigt die Ergebnisse der resultierenden Kampagnen. Um die Vergleichbarkeit der Versuchsreihen zu ermöglichen, sind die Werte nunmehr in Prozent angegeben. Wie erwartet, zeigen sich auch im Falle der Mehrfachbitfehler keine unerkannten Datenfehler für den Codierungsschlüssel $A = 58\,659$ mit $d_h = 6$. Mit der Fehlerzahl sinkt jedoch auch die Abdeckung des Fehlerraums von anfänglich 16,13 Prozent auf 0,04 Prozent bei den 5-fach-Bitfehlern ab. Entsprechend eignen sich diese statistischen Experimente lediglich als Indikator für das erwartete Verhalten, jedoch nicht als Nachweis für die Abwesenheit von unerkannten Datenfehlern.

In der Praxis angekommen

Insgesamt verhält sich der CoRed-Mehrheitsentscheider auch in der konkreten Umsetzung entwurfsgemäß. Die Fehlerinjektion zeigt die *vollständige Fehlererfassung für Einzel- und Doppelbitfehler*. Zudem bestätigen die Ergebnisse der Fehlerinjektion den in der Fehlersimulation gezeigten Einfluss der Codierungsparameter. Die Evaluation ist jedoch plattformspezifisch und muss entsprechend bei einem Wechsel der Ausführungsumgebung wiederholt werden.

Mehrbitfehler in Registern ($A = 58\,659$)		3 Bit	4 Bit	5 Bit
Ohne Auswirkung (verdeckter Defekt)		33,742 %	33,605 %	33,544 %
Erkannte Fehler	CoRed (arithmetische Codierung)	18,209 %	18,356 %	18,431 %
	Hardware Ausnahme	0,001 %	$<$0,001 %	0 %
	Schutzverletzung (Programmcode)	0,054 %	0,009 %	0,001 %
	Schutzverletzung (Speicher)	47,993 %	48,030 %	48,023 %
	Terminüberschreitung	0 %	0 %	0 %
Unerkannter Datenfehler		**0**	**0**	**0**
	$\sum$	579 838	627 886	$1{,}3 \cdot 10^6$
Größe des Fehlerraums		$3{,}59 \cdot 10^6$	$1{,}03 \cdot 10^8$	$2{,}90 \cdot 10^9$
Abdeckung		16,13 %	0,59 %	0,04 %

Tabelle 4.6: Erweiterte Fehlerinjektion von Mehrfachbitfehlern mit 3–5 Bit.
Die Ausdehnung der Experimente auf Mehrfachbitfehler stellt die Codierungseigenschaften des Super-As 58 659 auf die Probe. Entsprechend der ermittelten Hamming-Distanz von $d_h = 6$ treten keine unerkannten Datenfehler auf. Durch die explodierende Größe des Fehlerraums ist jedoch keine vollständige Abdeckung mehr möglich. In den Experimenten wurden daher zufällige Fehlermuster für die Injektion genutzt.

4.4 Diskussion der Ergebnisse

Die in Kapitel 4.2 durchgeführte Wahl der Codierungsparameter stellt ein generelles Problem bei der Umsetzung der arithmetischen Codierung dar. Die hier entwickelte Vorgehensweise einer zweistufigen Auswahl sowohl die experimentellen Ergebnisse der Distanzberechnung und Fehlersimulation lassen sich entsprechend auch auf andere Anwendungsbereiche der arithmetischen Codierung verallgemeinern. Dies gilt im selben Maß auch für die Behandlung der binären Darstellung der Codewörter.

4.4.1 Alternative Auswahlkriterien – Arithmetische Distanz

Das in Kapitel 4.2 herangezogene Konzept der Hamming-Distanz (d_h) (vgl. Exkurs, Seite 80) spielt im Allgemeinen eine entscheidende Rolle für die in der Datenkommunikation eingesetzte Codierungstheorie [Ham80]. Bis zu diesem Punkt konzentrierte sich die gesamte Betrachtung und Evaluation der Effektivität der Codierung daher auf die Hamming-Minimaldistanz der eingesetzten arithmetischen Codes. In der Literatur wird jedoch häufig die von Peterson [PW72] eingeführte *Arithmetische Distanz* (engl. *arithmetic distance*) d_a als ein geeigneteres Maß für die Betrachtung der arithmetischen Codierung von Berechnungen herangezogen [Rao74, MS09, SMM10, BGM12]. Die Distanz entspricht wiederum der für die Mutation eines gültigen Codewortes in ein anderes gültiges Codewort notwendigen Fehlerzahl. Im Unterschied zum Hamming-Abstand leitet sich die arithmetische Distanz jedoch aus der dezimalen anstelle der binären Darstellung der Codewörter ab. Das Ziel dieser geänderten Betrachtungsweise ist die explizite Berücksichtigung von Berechnungen beziehungsweise deren Auswirkungen auf den Abstand zwischen Codewörtern. Abbildung 4.7 veranschaulicht diesen Zusammenhang am Beispiel einer Addition. Ein Einzelbitfehler in den Eingabedaten kann sich durch die bei der Addition auftretenden Überträge (engl. *carry*) zwischen Bitpositionen zu einem Mehrfachbitfehler in der entstehenden Summe entwickeln.

Die arithmetische Distanz modelliert daher das Verhalten eines binären Addierers, indem sie zusätzlich den Übertrag berücksichtigt. Ein Fehler wird entsprechend als Addition beziehungsweise Subtraktion einer Zweierpotenz aufgefasst. Der Abstand zwischen zwei Codewörtern entspricht in

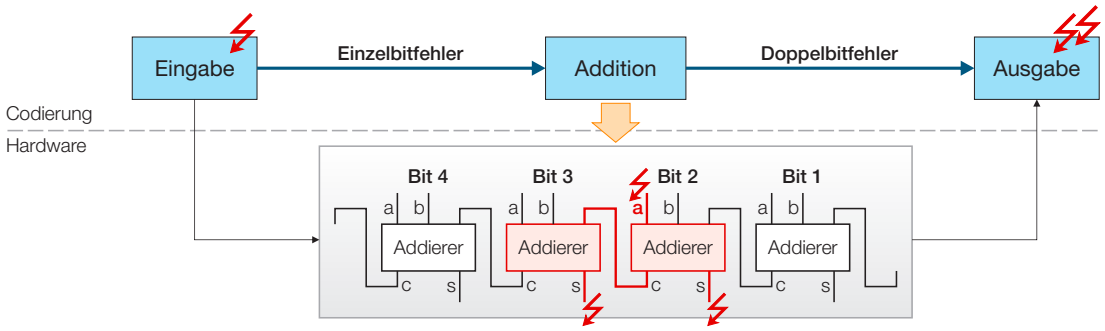


Abbildung 4.7: Fehlerbetrachtung am Beispiel einer codierten Addition.
Für die Bewertung der Effektivität der Codierung lassen sich unterschiedliche Sichtweisen einnehmen. Die *Hamming-Distanz* beschreibt die maximal erkennbare Fehlerzahl am Ende der Berechnung und der *Ausgabe* des Ergebnisses – die Auswirkungen der Addition bleiben hierbei unberücksichtigt. Die Fehlerzahl kann jedoch beim Durchlaufen der Hardware verändert werden. In diesem Beispiel wird ein zweiter Bitfehler durch einen Übertrag im Addierer hervorgerufen. Die *arithmetische Distanz* [PW72] modelliert daher das Verhalten eines binären Addierers und erlaubt in diesem speziellen Fall die Fehlerbetrachtung am *Eingang*.

der Folge der minimalen Anzahl fehlerhafter Ausgänge des Addierers, die für eine unerkannte Mutation erforderlich sind. Abstrakt betrachtet, entsprechen die fehlerhaften Ausgänge Termen der Form $\pm 2^i$ [PW72]. Ausgehend von den Codewörtern $\{6\ \texttt{[0110]}_2, 9\ \texttt{[1001]}_2\}$ ergibt sich beispielsweise eine arithmetische Distanz von $d_a = 2$ (zwei Terme genügen: $6 + 2^1 + 2^0 = 9$), wohingegen sich der Hamming-Abstand auf $d_h = 4$ beläuft. Ein wesentlicher Vorteil der arithmetischen Distanz ist die Möglichkeit der pessimistischen Abschätzung der Distanz für einen bestimmten Code [Man67, Mas64]. Der Berechnungsaufwand ist hierbei linear zu der Anzahl der Codewörter, im Gegensatz dazu nimmt dieser bei der Hamming-Distanz quadratisch zu.

Grundsätzlich verlagert sich durch diese Art der Fehlermodellierung die Betrachtungsebene von der reinen Codierung hin zu den Eigenschaften der ausführenden Hardwareeinheiten. Die arithmetische Distanz beschreibt daher letztlich die maximale Anzahl der Fehler die am *Eingang* eines binären Addierers vorliegen dürfen, um die Integrität des Codes im Ergebnis nicht zu gefährden. Im Gegensatz dazu charakterisiert die Hamming-Distanz die Anzahl der Fehler, die am *Ausgang* vorhanden sein dürfen, um noch eine sichere Fehlererkennung zu gewährleisten. Auf den ersten Blick vereinfacht die arithmetische Distanz die Wahl geeigneter Codierungsparameter, da sie die Auswirkungen der Berechnungsschritte berücksichtigt. Allerdings beschränkt sich das zugrundeliegende Modell auf eine simple binäre Addition mit Übertrag (engl. *ripple-carry adder*). Es existieren darüber hinaus jedoch noch eine ganze Reihe weiterer Konzepte für die Umsetzung von Addierern in der Hardware (z. B. *carry-select* oder *carry-lookahead* Addierer), welche ein anderes Verhalten aufweisen. Zudem setzen sich Berechnungen üblicherweise aus einer Vielzahl von verschiedenen arithmetischen und logischen Operationen zusammen, sie durchlaufen dementsprechend nicht nur die Addierer des Prozessors. In der Praxis lässt sich die Fehlerkette von der Codierung eines Wertes bis zu seiner Decodierung kaum adäquat modellieren.

Insgesamt erscheint daher der Einsatz der arithmetischen Distanz für die Bewertung der Robustheit der Codierung abseits der modellierten Addierer als wenig zielführend. Die auf der Hamming-Distanz basierende Fehlerbetrachtung des *Endprodukts* ist wesentlich leichter zu erfassen und unabhängig von den spezifischen Eigenschaften der Verarbeitungseinheiten der Hardware einsetzbar. Zudem ist die Hamming-Distanz nicht pessimistisch, sondern erlaubt exakte Aussagen über die erreichbare Effektivität der Codierung zum Zeitpunkt der Decodierung. Zu beachten ist jedoch, dass keine Rückschlüsse auf die maximale Fehlerzahl an bestimmten Punkten der Berechnungskette möglich sind.

4.4.2 Aussagekraft und Übertragbarkeit der Fehlerinjektion

Entgegen der allgemeinen Auswahl der Codierungsparameter gestaltet sich eine Übertragung der Evaluationsergebnisse bei der Fehlerinjektion auf der Befehlssatzebene schwierig. Diese hängen zum einen maßgeblich von der Darstellung des Binärprogramms und zum anderen von dem konkreten Verhalten der Ausführungsumgebung ab.

Fehlerinjektion auf der Befehlssatzebene

Analog zu der erweiterten Fehlermodellierung der arithmetischen Distanz, steht auch die Eignung der Fehlerinjektion auf der Befehlssatzebene hinsichtlich der Aussagekraft der Ergebnisse in der Diskussion [KKA93, KIR+99, MKT+11, CMC+13]. Wiederum wird der fehlende Zusammenhang zwischen dem durch den transienten Fehler initial hervorgerufenen Defekt in der Hardware und dem auf der Befehlssatzebene sichtbaren Fehlermuster angemerkt. Ein Einzelbitfehler in einer Logikeinheit kann sich beispielsweise zu einem Mehrfachbitfehler in einem Register entwickeln.

Das dem CoRed-Ansatz zugrundeliegende Entwurfsprinzip trifft jedoch keine Annahmen zu der eigentlichen Fehlerursache, sondern konzentriert sich vielmehr auf die endgültigen Auswirkungen am Ausgang. Die Wahl der Codierungsparameter erfolgt daher mit dem Ziel einer *bestmöglichen* Fehlererkennung, selbst bei einer Überbeanspruchung der Auslegung. Letztlich zeigen die in der erweiterten Fehlerhypothese in Kapitel 4.1 genannten empirischen Studien trotz der komplexen Zusammenhänge einen überwiegenden Anteil an Einzel- und Doppelbitfehlern, welche im Rahmen der Fehlerinjektion vollständig evaluiert werden konnten.

Eingeschränkter Parameterraum der Fehlerinjektion

Eine vereinfachende Annahme der durchgeführten Fehlerinjektion ist die Vernachlässigung des Parameterraumes für die eigentlichen Werte (v), da dieser Teil bereits durch die Fehlersimulation abgedeckt wird. Aufgrund der Einzelfehlerannahme lassen sich Datenflüsse und Kontrollflüsse unabhängig voneinander betrachten. Daher bleibt der sich lediglich auf den Datenfluss auswirkende Wert v im Rahmen der Fehlerinjektion fest und wird entsprechend nicht zu dem entstehenden Fehlerraum hinzugerechnet. Insgesamt zeigen die Experimente zur Überprüfung der Hamming-Minimaldistanz sowie zur Restfehlerwahrscheinlichkeit auf der Befehlssatzebene keine Anhaltspunkte für eine mögliche Abweichung zwischen Fehlersimulation und Fehlerinjektion in dieser Hinsicht.

Wechsel der Ausführungsplattform

Die Experimente der Fehlerinjektion ermöglichen in ihrer derzeitigen Form keine generelle Validierung des CoRed-Mehrheitsentscheiders. Vielmehr handelt es sich hierbei um eine exakte, jedoch plattformspezifische Verifikation des entsprechenden Binärprogramms. Die in Kapitel 4.3.3 beschriebene Schwachstelle der Isolation ist ein Beispiel für eine spezifische Eigenschaft der Ausführungsumgebung, welche nur unter bestimmten Umständen zutage tritt. Ebenso sind weitere stille Annahmen und Spezifika anderer Plattformen nicht auszuschließen. Bei einem Wechsel der Hardware, des Betriebssystem oder des Compilers ist eine erneute Evaluation daher unumgänglich. Die entkoppelte Betrachtung der Codierungsparameter reduziert jedoch den allgemeinen Aufwand bereits erheblich. Die Experimente zur Bestimmung der Hamming-Minimaldistanz sowie der Restfehlerwahrscheinlichkeit sind vollkommen unabhängig von der Ausführungsplattform wiederverwendbar. Es bleibt letztlich die vollständige Fehlerinjektion des CoRed-Mehrheitsentscheiders, welche aufgrund der vergleichsweise kompakten Größe und der Verfügbarkeit von Werkzeugen wie FAIL* relativ einfach durchführbar ist.

4.4.3 Arithmetische Codierung ganzer Anwendungen

Der überwiegende Teil dieses Kapitels befasst sich mit der Verbesserung der arithmetischen Codierung und deren entwurfsgemäßem Verhalten. Durch die geeignete Wahl der Codierungsparameter und die Vermeidung von Schwachstellen in der technischen Umsetzung erweist sich die Fehlererkennung der arithmetischen Codierung in CoRed als vollständig und zuverlässig.

Grundsätzlich sind diese Erkenntnisse nicht auf die Anwendung im Rahmen des CoRed-Ansatzes beschränkt und lassen sich auch auf andere Anwendungsformen der arithmetischen Codierung übertragen. Anstelle der von CoRed verfolgten Kombination von Codierung und Replikation ist beispielsweise die vollständige arithmetische Codierung der gesamten sicherheitskritischen Anwendung denkbar, wie sie teilweise von den in Kapitel 2.2.2 diskutierten Arbeiten verfolgt wird.

Kosten

Dem entgegen stehen zunächst die hierdurch verursachten Kosten. Selbst die vergleichsweise stark optimierte Umsetzung des CoRed-Mehrheitsentscheiders weist eine im Vergleich mit der uncodierten Variante annähernd dreifache Größe auf. Hinzu kommen die Kosten für die Codierung und Decodierung der Daten in den Replikaten. Im Allgemeinen zeigen die auf der vollständigen ANBD-Codierung der Anwendung bauenden Ansätze, wie zum Beispiel Software Encoded Processing (SEP) [WF07] oder Compiler Encoded Processing (CEP) [SSSF10, Sch11b], eine anwendungsabhängige Steigerung der Laufzeitkosten in der Größenordnung von 10^2 bis 10^5.

CoRed erkauft sich die Absicherung der Anwendungskomponenten zwar durch deren replizierte Ausführung, die hiermit verbundenen Kosten liegen mit näherungsweise dem Faktor drei jedoch deutlich unterhalb der vollständigen Codierung der Anwendung mittels der eben genannten Ansätze.

Fehlererkennung und Fehlerdiagnose

Einen weiteren wesentlichen Aspekt stellt die Fehlererkennung und Fehlerdiagnose dar. Im Zuge der arithmetischen Codierung lassen sich Fehler lediglich im Rahmen der Decodierung beziehungsweise expliziten Überprüfung der Codewörter aufdecken. Bei einer vollständigen Codierung der Anwendung ist die Fehlererkennung innerhalb der Ausführung auf dedizierte Überprüfungspunkte angewiesen. Ohne eine entsprechend feingranulare Prüfung ist zudem die Fehlerdiagnose und Einschränkung des Fehlerbereichs nicht möglich. Die eben erwähnten Ansätze nutzen hierfür eine externe Prüfinstanz der Ausführungsumgebung. Diese induziert zum einen weitere Kosten und stellt darüber hinaus eine kritische Fehlerstelle dar.

Mit Blick auf die mögliche Veränderung der Bitfehlerzahl (vgl. Kapitel 4.4.1) bei der Durchführung von Berechnungen auf Codewörtern wird hierdurch zudem die kritische Frage nach einem geeigneten Abstand zwischen den Prüfpunkten aufgeworfen: Da sowohl die Berechnungsfolge als auch der Einfluss der einzelnen Berechnungsschritte auf das Fehlermuster anwendungsabhängig sind, ist prinzipiell eine hohe Bandbreite für die Fehlererkennung erforderlich. Für die Kontrollflussüberwachung in CEP [SSSF10] erfolgt die Überprüfung beispielsweise beim Betreten beziehungsweise Verlassen jedes Basisblocks – mit einem entsprechend negativen Einfluss auf die Kosten.

Demgegenüber setzt CoRed die Codierung sparsam und unabhängig von der Anwendung für eine feste Zahl an Infrastrukturfunktionen wie den CoRed-Mehrheitsentscheider ein. Die Umsetzung ist durch ihren statischen Charakter zum einen sehr gut testbar und beschränkt zum anderen die Zahl der Berechnungsschritte zwischen der Codierung und Decodierung durch ihre geringe Größe auf einige wenige Operationen. Zusammen mit der von der Schwere der Bitfehler unab-

hängigen und vollständigen Fehlererfassung der replizierten Ausführung der Anwendung ergeben sich somit insgesamt bessere Eigenschaften bei niedrigeren Kosten und reduzierter Komplexität.

4.5 Zusammenfassung

Der Weg vom theoretischen Entwurf hin zu einer praktischen Realisierung der arithmetischen Codierung im Allgemeinen sowie des CoRed-Mehrheitsentscheiders im Speziellen ist mit einer ganzen Reihe von Problemen und technischen Herausforderungen verbunden. Grundsätzlich können sowohl eine ungünstige Parametrierung der arithmetischen Codierung als auch mögliche Schwachstellen in der technischen Umsetzung die tatsächlich erreichbare Effektivität der Fehlererkennung signifikant schwächen. In der Folge können unerkannte Datenfehler trotz einer konzeptionell vollständigen Fehlererkennung auftreten.

Ein erster Schritt zur Vermeidung dieser Probleme ist die geeignete *Wahl der Codierungsparameter*. In der Literatur finden sich hierzu lediglich wage Empfehlung, wie zum Beispiel der Einsatz von großen Primzahlen. Die in diesem Kapitel durchgeführte experimentelle Evaluation aller durch 16-Bit-Schlüssel darstellbaren Codes zeigt jedoch eindrucksvoll, dass *je größer, desto besser* nicht zwangsläufig gilt. CoRed nutzt stattdessen einen zweistufigen Prozess, welcher die möglichen Codierungsschlüssel zunächst nach ihrer absoluten Erkennungsleistung und der minimalen Hamming-Distanz klassifiziert. Darauf aufbauend erlaubt die Fehlersimulation die Bestimmung der Restfehlerwahrscheinlichkeit bei einer Überbeanspruchung der Codierung und somit die weitere Optimierung der Parameter. Hierdurch lassen sich die Schlüssel zusätzlich in *gutmütig* und *tückisch* einteilen. Letztere überschreiten in bestimmten Fehlerszenarien die vorhergesagte Restfehlerwahrscheinlichkeit teilweise deutlich und sind daher weniger vorhersagbar. Insgesamt konnten für die untersuchten 32-Bit-AN-Codes fünf besonders leistungsfähige Super-*A*s identifiziert werden – keine davon eine Primzahl. Diese erlauben die sichere Erkennung von bis zu 5-fach-Bitfehler, wobei zusätzlich drei von ihnen ein gutmütiges Verhalten bei Überbeanspruchung aufweisen.

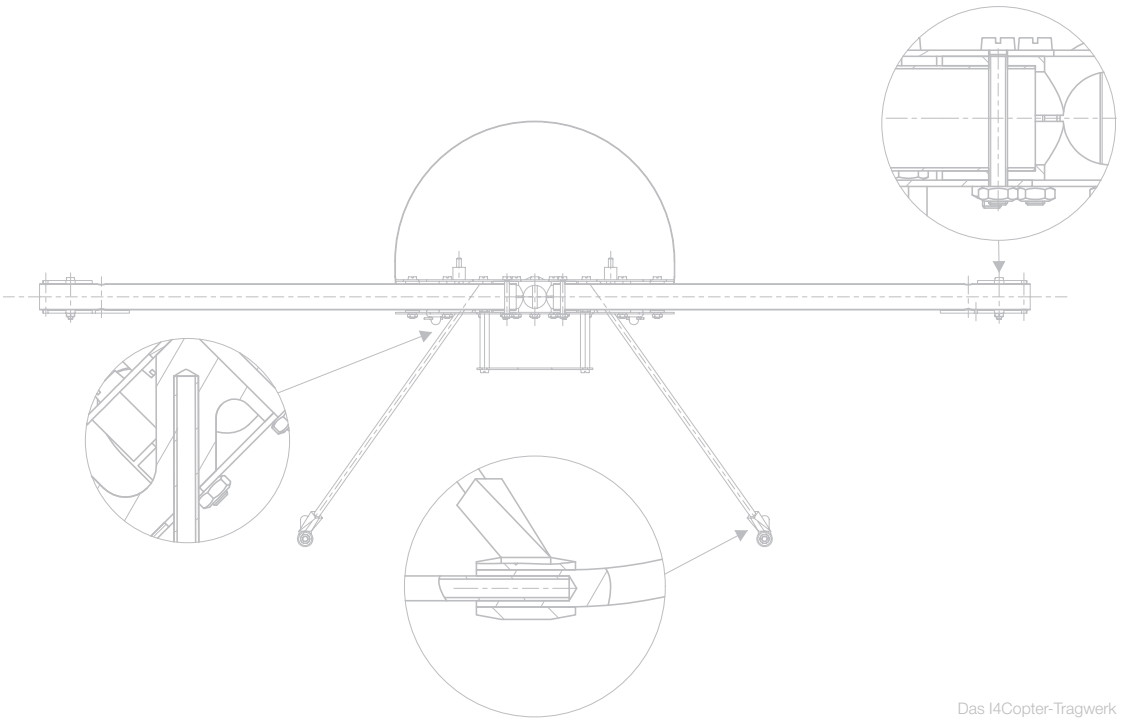
Darüber hinaus stellen *Schwachstellen in der Implementierung* eine weitere Quelle für einen signifikanten Verlust der Erkennungsleistung dar. Dies umfasst dabei sowohl Annahmen hinsichtlich der Abgeschlossenheit des Coderaums durch die Codierungstheorie als auch die Ausnutzung bestimmter Konventionen der Binärschnittstelle der Hardwareplattform durch den Compiler und das Betriebssystem. Die entstehenden Schwachstellen treten ausschließlich im Fehlerfall zutage und stellen dabei üblicherweise Grenzfälle dar. Ihre Aufdeckung erfordert daher eine vollständige Fehlerinjektion des gesamten Fehlerraums auf der Befehlssatzebene und damit letztlich den Nachweis der Übereinstimmung zwischen Entwurf und Binärprogramm.

Die Evaluationsergebnisse zeigen, dass auch die technische Umsetzung von CoRed in der Lage ist, die arithmetische Codierung bis an ihre Leistungsgrenze zu führen und darüber hinaus alle verbleibenden Schwachstellen zu beseitigen. Im Vergleich zu gängigen Ansätzen zeigt sich dies in einer deutlichen Verbesserung der Erkennungsleistung – wie das Beispiel des Codierungsschlüssels $A = 251$ und einer Senkung der Restfehlerwahrscheinlichkeit um den Faktor 1000 gezeigt hat. Zusammenfassend ermöglicht die technische Umsetzung von CoRed die *vollständige* und *zuverlässige* Fehlererkennung, wie sie durch den Entwurf vorgesehen ist.

Teil II

Anwendungsebene

Replikation von Regelungsanwendungen



Das I4Copter-Tragwerk

5

Fallbeispiel – I4Copter

Fluglageregelung eines unbemannten Luftfahrzeugs

Im ersten Teil dieser Arbeit wurden mit CoRed die Bausteine für die gezielte Beeinflussung der nicht-funktionalen Eigenschaft *Zuverlässigkeit* auf Ebene des Betriebssystems entwickelt. Die Fehlertoleranz basiert hierbei auf einer Kombination aus einer replizierten Ausführung der Prozessinkarnationen der Anwendung sowie der arithmetischen Codierung der eigentlichen Replikationsinfrastruktur. CoRed kommt dabei im Allgemeinen ohne funktionale Änderungen der Anwendung aus und kann entsprechend transparent eingesetzt werden. Dies gilt im Besonderen für die Klasse der Regelungsanwendungen, welche durch die hier vorherrschende Eingabe-Verarbeitung-Ausgabe-Struktur sowohl eindeutige Schnittstellen als auch einen unterbrechungsfreien Durchlauf der Ausführung aufweisen.

Im zweiten Teil der Arbeit verlagert sich nun der Fokus von der System- auf die Applikationsebene und die Anwendung des CoRed-Ansatzes auf die angestrebte Zieldomäne der Regelungsanwendungen. Vor der allgemeinen Betrachtung dieser Thematik in Kapitel 6, veranschaulicht dieses Kapitel die grundsätzliche Umsetzbarkeit der CoRed-Entwurfsmuster mithilfe eines konkreten Fallbeispiels. Dieses bezieht sich auf die Fluglageregelung des I4*Copter*s [UKH+11, UFH+12], einem im Rahmen dieser Arbeit entwickelten beziehungsweise erweiterten unbemannten Luftfahrzeug (engl. *unmanned aerial vehicle*) (UAV). Aus Anwendungssicht handelt es sich hierbei um ein Mehrgrößensystem (engl. *multiple Input, multiple Output*) mit einer umfangreichen Sensorik und Aktorik, dessen regelungstechnische Behandlung ein anspruchsvolles und realitätsnahes Beispiel darstellt. Darüber hinaus weist die Regelungsanwendung ein ausgewiesenes Schutzbedürfnis gegenüber transienten Fehlern auf: Die Auswirkungen eines Fehlverhaltens sind in jedem Fall schwerwiegend und können bis zum Verlust des Luftfahrzeugs führen.

Das Ziel dieses Kapitels ist daher die Anwendung des CoRed-Ansatzes auf die sicherheitskritische Fluglageregelung des I4*Copter*s. Hierfür führt Kapitel 5.1 zunächst in den grundlegenden technischen Aufbau des I4*Copter*s sowie dessen Softwaresystem ein. Darauf aufbauend erfolgt in Kapitel 5.2 die Replikation der eigentlichen Regelungsanwendung und eine damit einhergehende erneute experimentelle Verifikation der Effektivität von CoRed durch eine Fehlerinjektionskampagne auf der konkreten Hardwareplattform. Abschließend wirft Kapitel 5.3 einen Blick auf die in

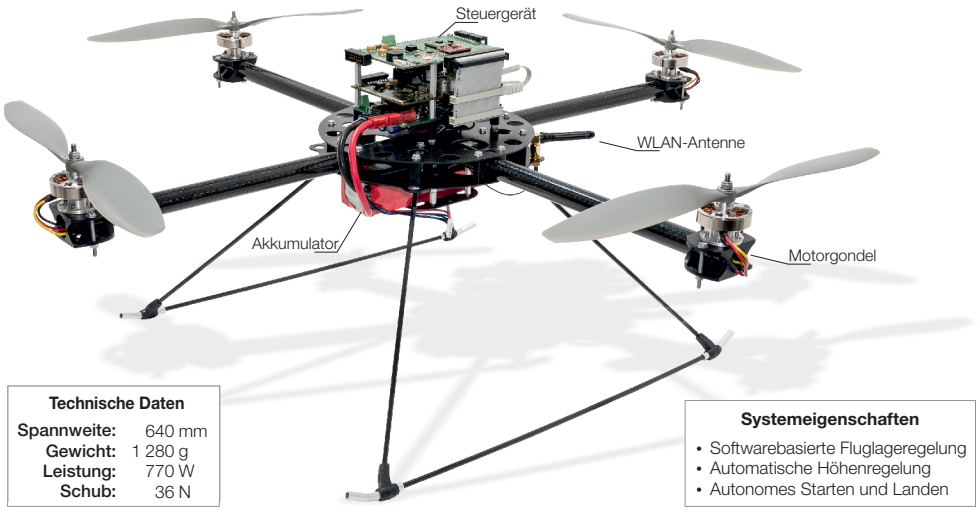


Abbildung 5.1: CoRed Fallstudie – Der I4*Copter* Demonstrator für sicherheitskritische eingebettete Systeme.
Der I4*Copter* ist ein senkrecht startendes unbemanntes Luftfahrzeug. In der Konfiguration mit vier Propellern wird dieser häufig *Quadrocopter* genannt. Bei diesem Typ ist durch das inhärent instabile Flugverhalten eine kontinuierliche Regelung der Fluglage unumgänglich. Diese erfolgt beim I4*Copter* rein softwarebasiert durch eine entsprechend *sicherheitskritische Fluglageregelung*. In der Mitte befindet sich das zentrale Steuergerät aus Rechensystem, Sensorik und Leistungselektronik (vgl. Abbildung 5.2).

Kapitel 3.4 aufgezeigte Erweiterbarkeit der Fehlerkennung. Hierfür wird der Redundanzbereich der Fluglageregelung über die Systemgrenzen hinaus auf die Aktorik und Sensorik ausgedehnt.

5.1 Das I4*Copter* Luftfahrzeug

Der I4*Copter* ist ein senkrecht startendes und landendes unbemanntes Fluggerät. In der in Abbildung 5.1 dargestellten Konfiguration mit vier Propellern werden diese Luftfahrzeuge häufig auch als *Quadrocopter* (engl. *quadrotor*) bezeichnet. Sie zählen wie die Hubschrauber zur Klasse der Drehflügler, besitzen jedoch feststehende Propeller. Eine Änderung der Fluglage erfolgt daher alleine durch die geeignete Anpassung der Drehzahl der Motoren. Im Gegensatz zu den meisten anderen Luftfahrzeugen weisen Quadrocopter daher inhärent instabile Flugeigenschaften auf, was eine kontinuierliche Regelung der Fluglage unumgänglich macht. Dies erfordert zunächst die Erfassung der Lage im Raum durch eine geeignete Sensorik. Darauf aufbauend muss die Fluglage durch eine adäquate Ansteuerung der Motoren stabilisiert werden, um letztlich das gewünschte Flugverhalten zu erzielen.

Der technische Aufbau des I4*Copter*s orientiert sich aus Gründen der Praxisnähe an der in der Fahrzeugtechnik üblichen Hardware. Darüber hinaus wurde die zentrale Regelungsanwendung in enger Zusammenarbeit mit erfahrenen Regelungstechnikern und nach gängigen Entwurfsmethoden entwickelt und rein softwarebasiert umgesetzt [UFH+12]. Im Rahmen dieser Arbeit dient der I4*Copter* daher als anspruchsvolles Fallbeispiel für eine sicherheitskritische Regelungsanwendung.

Zu den zentralen Eigenschaften der Regelungsanwendung zählt die obligatorische *Fluglageregelung* bei manueller Steuerung. Darüber hinaus ermöglicht die Anwendung auch ein *autonomes Starten* und *Landen* in Bodennähe sowie die *Regelung der Flughöhe*. Die Konstruktion der Flugzelle und des Antriebsstrangs richtet sich weitgehend nach den Abmessungen des zentralen Steuergerätes. Die sich hieraus ergebenden technischen Daten liegen mit circa 1,3 Kilogramm Gewicht und

64 Zentimeter Spannweite noch im typischen Modellbaubereich.

Für ein besseres Verständnis der zu schützenden Fluglageregelung gibt der folgende Abschnitt zunächst einen groben Überblick über den Aufbau und die Struktur des Hardware- und Softwaresystems des I4*Copter*s sowie der eingesetzten Sensorik und Aktorik. Für eine darüber hinaus gehende, detaillierte Beschreibung des Systems sei an dieser Stelle auf die zugehörigen Veröffentlichungen [UKH+11, UHK+12, UFH+12] beziehungsweise studentischen Abschlussarbeiten [Dre08, HK08, Fra09, Sch10, Lim11, Bla11, Hei11, Dre11, Fra11, Reb12, Kla12] verwiesen. Die beiden Exzerpte auf den Seiten 110 und 112 erlauben eine direkte Abkürzung zur Anwendung von CoRed auf die Fluglageregelung in Kapitel 5.2 ab Seite 112.

5.1.1 Hardwaretechnischer Aufbau

Das Steuergerät des I4*Copter*s besteht analog zum bekannten Eingabe-Verarbeitung-Ausgabe-Schema aus den in Abbildung 5.2 dargestellten drei Platinen für die Erfassung der Umweltparameter, die Berechnung der notwendigen Regelungsaufgaben sowie den Betrieb der Antriebseinheiten. Sowohl die Sensor- als auch die Motorplatine wurden hierbei eigens für den spezifischen Anwendungszweck entwickelt. Zusammengenommen haben die eingesetzte Sensorik und Aktorik einen signifikanten Einfluss auf den Entwurf der eigentlichen Regelungsanwendung. Im Folgenden wird ihr Aufbau daher beginnend mit der Inertial-Messeinheit näher betrachtet.

Sensorplatine – Inertial-Messeinheit

Die Grundvoraussetzung für eine erfolgreiche Regelung der Fluglage ist die geeignete Erfassung der Bewegung des Fahrzeugs im Raum. Da eine direkte Bestimmung der absoluten Lage eine äußere Referenz erfordert [PMC06], kommt im Falle des I4*Copter*s eine *Inertial-Messeinheit* (engl. *inertial measurement unit*) (IMU) für die relative Lagebestimmung zum Einsatz.

Die Inertial-Sensorik überwacht hierfür die sechs möglichen Freiheitsgrade des Flugobjekts, bestehend aus der *Bewegung* beziehungsweise *Drehung* um die drei Raumachsen. Die in Abbildung 5.2 (oben) dargestellte und spezifisch für das I4*Copter*-System entwickelte *Sensorplatine* stellt die notwendigen Messgrößen zur Verfügung. Diese umfassen in erster Linie die obligatorische Drehrate und Beschleunigung um die drei Achsen. Darüber hinaus stehen Sensoren für die Messung der relativen und absoluten Höhe über Grund sowie der magnetischen Flussdichte zur Verfügung – letztere erfüllen die Funktion eines Kompass.

Wie bereits in Kapitel 3.4.1 erläutert, weisen die technischen Bauteile der Sensorik im Vergleich zu dem eigentlichen Rechensystem in der Regel eine deutlich niedrigere Zuverlässigkeit auf [Fra10]. Unabhängig von der Fehlerrate für transiente Fehler des Rechensystems nutzen sicherheitskritische Systeme daher üblicherweise eine strukturell redundante Sensorik. Diesem Umstand trägt die Auslegung der I4*Copter*-Sensorplatine durch die Bereitstellung von insgesamt zwölf Sensorbausteinen Rechnung. Die kritische Messung der Drehrate und Beschleunigung ist hierbei dreifach, die der Höhe und magnetischen Flussdichte jeweils zweifach redundant ausgelegt. In der Summe liefern die Sensoren 35 dedizierte Messwerte für die Überwachung der sechs Freiheitsgrade.

Um für das Fallbeispiel eine möglichst breite Spanne an Systemfunktionen und Treibern des Rechensystems abzudecken, sind die Sensoren einzeln und in der Regel für jede physikalische Messgröße über unterschiedliche Schnittstellen mit diesem verbunden. Abhängig von dem konkreten Sensorbaustein erfolgt die Darstellung der Messwerte dabei entweder analog, digital oder binärcodiert. Insgesamt deckt die Inertial-Messeinheit des I4*Copter*s somit ein breites und realitätsnahes Spektrum an möglichen Eingangsquellen und Schnittstellen ab.

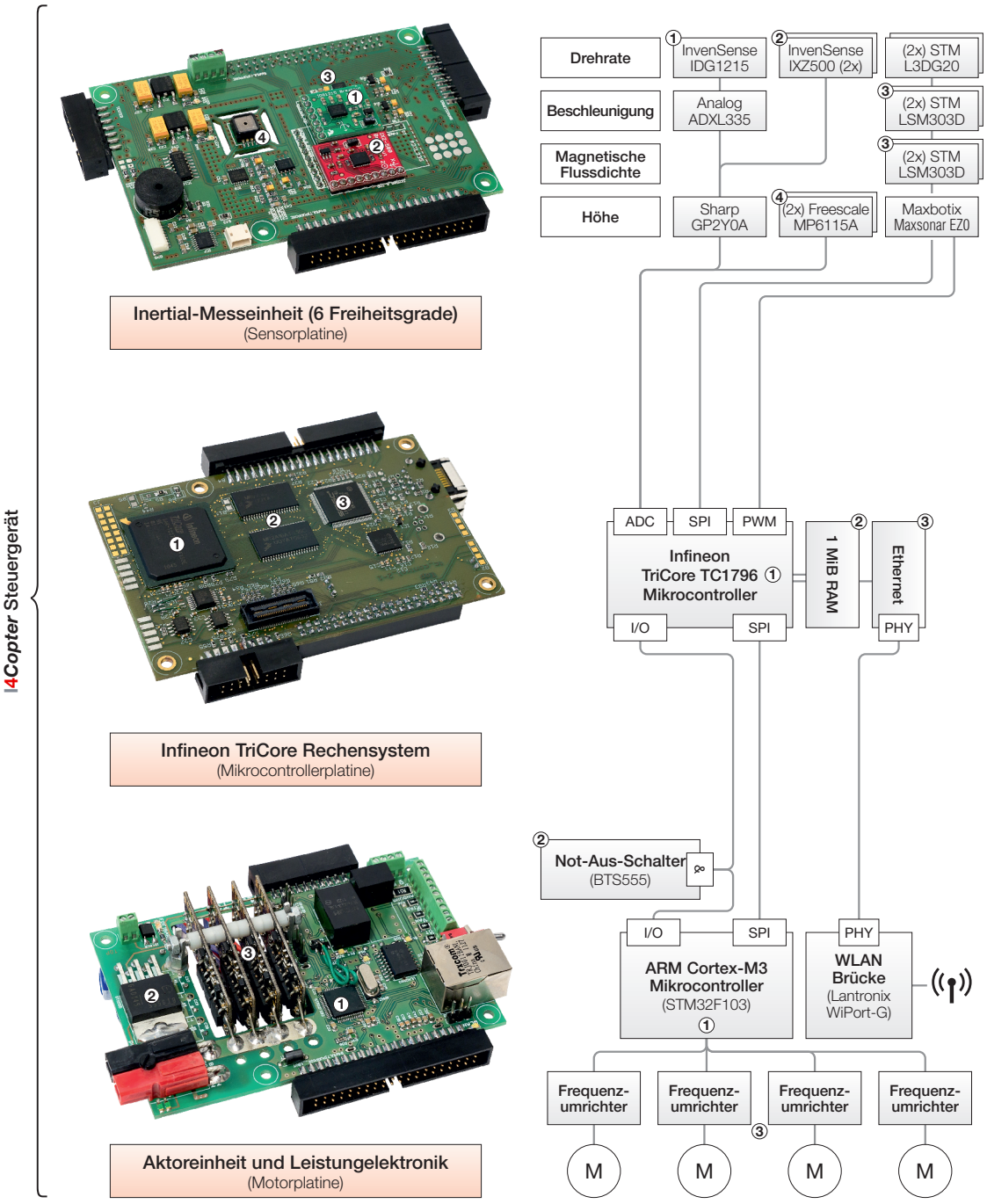


Abbildung 5.2: I4*Copter*-Steuergerät – Redundante Sensorik, TriCore-Mikrocontroller, Leistungselektronik und Aktorik. Das Steuergerät des I4*Copter*s setzt sich analog zu dem Eingabe-Verarbeitung-Ausgabe-Schema aus drei Stufen zusammen. Die speziell entwickelte *Sensorplatine* umfasst insgesamt zwölf Inertial-Sensoren für die redundante Erfassung der Lage im Raum. Ihre Messdaten werden auf der zentralen *Mikrocontrollerplatine* durch das TriCore-TC1796-Rechensystem verarbeitet und in Stellwerte für die vier Motoren umgesetzt. Die leistungstechnische Antriebssteuerung übernimmt die ebenfalls spezifisch ausgelegte *Motorplatine*, welche zu diesem Zweck über ein weiteres Rechensystem in Form eines ARM Cortex-M3 verfügt. Die Platinen sind über analoge und digitale Signalleitungen sowie serielle Kommunikationsbusse miteinander verbunden.

Mikrocontrollerplatine – TriCore Rechensystem

Das zentrale Rechensystem des I4*Copter*s basiert auf einem 32-Bit *Infineon TriCore TC1796 Mikrocontroller* [Inf07]. Die TriCore-Familie ist speziell für den Einsatz in der Fahrzeugelektronik ausgelegt und findet beispielsweise in der Fahrwerks- und Motorsteuerung Anwendung. Der hier verwendete TC1796 bietet daher unter anderem eine Speicherschutzeinheit sowie eine Vielzahl von Schnittstellen für die Anbindung von Sensorik und Aktorik. Mit Blick auf die Fehlerraten für transiente Fehler sind öffentlich bislang keine quantitativen Angaben verfügbar. Technisch ist jedoch lediglich der interne Arbeitsspeicher des Rechensystems durch Paritätsbits geschützt. Darüber hinaus verweist der Hersteller bei sicherheitskritischen Anwendungen auf die Notwendigkeit zusätzlicher Maßnahmen auf Anwendungsebene [HG13]. Der TC1796 stellt somit eine praxisnahe Plattform für den Einsatz des CoRed-Ansatzes dar. Die in Abbildung 5.2 (Mitte) dargestellte *Mikrocontrollerplatine* basiert auf der kommerziellen EasyRun-Evaluationsplatine von HighTec [Hig07] für den TriCore TC1796. Sie beherbergt neben dem eigentlichen Mikrocontroller ein Mebibyte externen Arbeitsspeicher sowie eine 100 $^{\text{Mbit}}/_{\text{s}}$ Ethernet-Schnittstelle. Letztere dient in Kombination mit einer WLAN-Brücke (engl. *wireless local area network*) auf der Motorplatine zur Kommunikation mit der Basisstation.

Da die Inertial-Sensorik, wie bereits beschrieben, über keine eigene Logik für die Abtastung der Messsignale verfügt, sind die Sensoren durch insgesamt 24 Signalleitungen mit der Mikrocontrollerplatine verbunden. Die Übertragung beziehungsweise Signalauswertung erfolgt aufgrund der diversitären Auslegung der Sensorik durch die *Analog-Digital-Umsetzer* (engl. *analog-to-digital converter, ADC*), *Binärzähler* (engl. *pulse-width modulation, PWM*) und *synchronen seriellen Datenbusse* (engl. *serial peripheral interface, SPI*) des TriCore Mikrocontrollers. Ebenso erfolgt die Ausgabe der Stellwerte an die Aktoreinheit durch einen weiteren SPI-Kommunikationsbus.

Für die im Rahmen dieses Kapitels durchgeführten Experimente wurde der Rechenkern des TriCores mit der höchstmöglichen Taktfrequenz von 150 MHz betrieben. Durch den internen Aufbau ergibt sich hieraus ein für die Peripherieeinheiten relevanter Systemtakt von 75 MHz. Darüber hinaus wurden die durch den Hersteller vorgegebenen Standardeinstellungen [Inf07] verwendet – wesentlich sind hierbei insbesondere die Aktivierung der Cache-Speicher (engl. *cache*) sowie der hardwareseitigen Fehlererkennung und Ausnahmen.

Motorplatine – Aktoreinheit und Leistungselektronik

Das Antriebssystem des I4*Copter*s stellt bis zu 770 Watt elektrischer Leistung respektive 36 Newton Schubkraft bereit. Die hierfür notwendige Leistungselektronik ist sowohl aus Gründen der elektromagnetischen Abschirmung als auch der Modularität als eigenständige Aktoreinheit ausgelegt. Den letzten Baustein des I4*Copter*-Steuergeräts stellt entsprechend die ebenfalls anwendungsspezifisch entwickelte *Motorplatine* dar. Sie dient in erster Linie der leistungstechnischen Steuerung der vier Antriebsmotoren, deren Phasen durch einen Frequenzumrichter elektronisch kommutiert werden müssen [Fis09]. Hierfür kommen die kommerziellen Leistungsregler Mikrokopter[1] BLControl v2 zum Einsatz. Daneben stellt die Aktoreinheit einen elektronischen Schalter für die Spannungsversorgung der Antriebsmotoren zur Verfügung. Dieser dient als *Not-Aus-Schalter* um das Fluggerät in Abhängigkeit von der Schwere einer Fehlersituation kontrolliert stillzulegen.

Die Motorplatine verhält sich dem zentralen TriCore-Rechensystem gegenüber als *intelligenter Aktor*, welcher seinerseits die Ansteuerung der einzelnen Frequenzumrichter übernimmt. Die Steuerung der Leistungselektronik erfolgt dabei durch ein eigenes Rechensystem in Form eines 32-Bit *ARM Cortex-M3 Mikrocontrollers*, welcher seine Stellwerte durch den SPI-Kommunikationsbus erhält. Diese Auslegung ermöglicht zum einen die in Kapitel 3.4.2 eingeführte Erweiterung

[1] Hersteller Webseite: `www.mikrokopter.de/ucwiki/BL-Ctrl_2.0`, Abruf: 13.03.2014

von CoRed auf der Ausgangsseite. Zum anderen lässt sich der I4*Copter* so um eine weitere Aktoreinheit zu einem Octocopter erweitern. In dieser Konfiguration werden je zwei Motoren koaxial an den Motorgondeln angeordnet, wodurch sich die schon auf der Sensorseite praktizierte Hardwareredundanz prinzipiell auch auf den Antriebsstrang ausdehnen lässt.

Das I4*Copter* System

Der I4*Copter* stellt eine praxisnahe Hardwareplattform dar: Das System basiert auf einem Infineon TriCore TC1796 Mikrocontroller. Diesem werden von der *Inertial-Sensorik* insgesamt 35 Messwerte über verschiedene Schnittstellen zur Verfügung gestellt. Die Ansteuerung der Aktoren erfolgt durch eine nachrichtenbasierte Stellwertübergabe an die *intelligente Leistungselektronik*, welche die geeignete Steuerung der Antriebsmotoren übernimmt.

5.1.2 Softwaresystem

Das Softwaresystem des I4*Copter*s orientiert sich naturgemäß an den Gegebenheiten und Anforderungen der Anwendung und des Hardwaresystems. Insgesamt umfasst das Softwaresystem für die Steuerung und Regelung des Fahrzeugs über 26 000 Zeilen C++-Quellcode. Dies beinhaltet neben der eigentlichen Regelungsanwendung eine ganze Reihe von Systemdiensten und Hardwaretreibern für die spezifischen Kommunikationsschnittstellen, Sensoren und Aktoren.

Als Ausführungsplattform für die Anwendungskomponenten dient das Forschungsbetriebssystem CiAO [LHSP+09]. Dieses implementiert das in der Fahrzeugtechnik verbreitete AUTOSAR OS-Applikationsmodell (engl. *AUTomotive Open System ARchitecture Operating System*) [AUT11] und unterstützt damit die erforderliche räumliche und zeitliche Isolation der Prozessinkarnationen mithilfe der Speicherschutzeinheit des TriCores. Die eigentliche Anwendung des I4*Copter*s setzt sich aus den folgenden drei Hauptkomponenten beziehungsweise Subsystemen zusammen: *Signalverarbeitung*, *Fluglageregelung* sowie *Steuerung und Kommunikation*.

Signalverarbeitung

Die Signalverarbeitung (`SignalProcessing`) dient der periodischen Erfassung der Umwelteigenschaften durch Abfrage der von der Sensorplatine bereitgestellten Messwerte. Hierfür nutzt die Signalverarbeitung die entsprechenden Treiber der Sensorbausteine. Diese lesen die Rohdaten in Form von Spannung, Pulslänge oder *serial peripheral interface (SPI)*-Nachricht von den jeweiligen Schnittstellen und Peripherieeinheiten des TriCores. Die erfassten Werte werden anschließend einer ebenfalls sensorspezifischen Signalvorverarbeitung zugeführt, welche die Umrechnung in die physikalischen Zieleinheiten und gegebenenfalls eine Vorfilterung der Signale durchführt. Mit dem Abschluss der Erfassung und Vorverarbeitung werden die redundanten Messwerte an die Sensorfusion weitergereicht und dort zu jeweils einem Messwert pro Messgröße und Freiheitsgrad verschmolzen. Die Sensorfusion ist hierbei bereits ein Teil des regelungstechnischen Entwurfs und wird entsprechend zusammen mit der eigentlichen Regelung durch das Entwurfswerkzeug MATLAB/SIMULINK generiert.

Die Häufigkeit der Datenfusion bestimmt sich typischerweise anhand der größten Bandbreite unter den eingesetzten Sensoren. Den begrenzenden Faktor bilden in diesem Zusammenhang die Gyroskopen mit analoger Schnittstelle und wenigstens 155 Hertz. Dem Abtasttheorem folgend ergibt sich hieraus im konkreten Fall eine Periode von drei Millisekunden für die Sensordatenerfassung und Verarbeitung. Entsprechend werden die Arbeitsaufträge für die Abtastung der Sensoren

und die Signalvorverarbeitung mit der Datenfusion zu einer Aufgabe verschmolzen und bilden gemeinsam eine Prozessinkarnation.

Fluglageregelung

Die Fluglageregelung (`FlightControl`) repräsentiert letztlich die eigentliche Regelungsanwendung des I4*Copter*s. Diese nutzt die vorverarbeiteten und fusionierten Messwerte zur Stabilisierung der Fluglage und zur Umsetzung der durch den Piloten vorgegebenen Führungsgrößen. Die errechneten Stellwerte der Regelung werden anschließend an das Aktorsystem übertragen. Aufgrund der Dynamik des Systems und der Vielzahl der Eingangsgrößen basiert der regelungstechnische Entwurf im I4*Copter* auf einer *Zustandsregelung* der Fluglage sowie der Flughöhe. Bei dieser Art des Entwurfs hängen die errechneten Stellwerte nicht alleine von den aktuellen Systemgrößen, wie zum Beispiel der Lage oder Drehrate, ab. Vielmehr werden die durch den Regler bestimmten Zustände geeignet auf die Stellwerte zurückgeführt und verrechnet. Vereinfacht ausgedrückt nutzt der Regler sein Wissen über die aktuelle Steuerung zur besseren Vorhersage des künftigen Systemzustands. Die Regelung erzielt hierdurch ein potenziell schnelleres und stabileres Führungsverhalten. Für die Umsetzung der Fluglageregelung ist entsprechend eine vollständige Kenntnis der Objektzustände notwendig. Im konkreten Fall lässt sich die Lage im Raum jedoch nicht direkt durch die gegebene Sensorik messen. Daher kommen *Beobachter* (engl. *observer*) für die Zustandsrekonstruktion der Drehrate, des Lagewinkels und der Flughöhe zum Einsatz. Der Entwurf von Reglern und Beobachtern erfolgte im konkreten Fall durch eine umfassende Modellierung und Analyse der Regelstrecke in MATLAB/SIMULINK. Die Erzeugung des entsprechenden C++-Quellcodes sowie die Bestimmung der Regelungsparameter erfolgte hierbei ebenfalls mithilfe des Entwurfswerkzeugs.

Die Fluglageregelung wird im I4*Copter* mit einer Periode von neun Millisekunden ausgeführt. Die zeitlichen Eigenschaften leiten sich hierbei in erster Linie aus der Objektdynamik ab und stützen sich auf die Trägheitsmomente des Fluggeräts und die Eigenschaften des Aktorsystems.

Steuerung und Kommunikation

Die Steuerung und Kommunikation (`CopterControl`) ist lediglich mittelbar an der Fluglage des I4*Copter*s beteiligt. Diese Komponente dient in erster Linie der Interaktion mit dem Piloten. Sie nutzt hierfür die Ethernet-Schnittstelle für die Kommunikation mit der Basisstation. Diese besteht aus einem mit der quelloffenen Fernsteuerungsanwendung QGROUNDCONTROL[2] ausgerüsteten Notebook. Die Eingaben des Piloten werden von dort über WLAN an den I4*Copter* übertragen. Nach dem Nachrichtenempfang werden die entpackten Sollwerte der Fluglageregelung zur Verfügung gestellt. In umgekehrter Richtung sendet `CopterControl` Telemetriedaten zurück an die Basisstation.

Die Kommunikation erfolgt bidirektional alle 21 Millisekunden, wobei sich die Basisstation grundsätzlich dem Takt des Fluggeräts anpassen kann. Diese Periodendauer ergibt sich zum einen als Harmonische aus der Periode der Fluglageregelung, zum anderen liegt die Zeitspanne zwischen den Steuersignalen damit ausreichend unter der Wahrnehmungsschwelle des Piloten.

Systemkomposition und Konfiguration

Die Anwendungskomponenten des Softwaresystems liegen ohne Ausnahme als C++-Klassen vor. Ihre Aufrufschnittstelle folgt dabei dem bekannten Muster aus Setzen der Eingaben (`set()`), Durchführen eines Berechnungsschritts (`step()`) und dem Abfragen der Ausgaben (`get()`). Für

[2]Projekt Webseite: `www.qgroundcontrol.org`, Abruf: 25.04.2014

die Abbildung der Anwendungsobjekte auf Prozessinkarnationen des Betriebssystems kommt eine generische *Hüllenklasse* (engl. *wrapper*) zum Einsatz. Diese hierfür genutzten C++-Templates sorgen zum einen für die statische Konfiguration und Instantiierung der Anwendungsklassen im Zuge des Übersetzungsvorgangs und zum anderen für eine saubere Kapselung des entstehenden Kontextes beziehungsweise dessen Zuordnung zu dedizierten Schutzdomänen. Hieraus ergibt sich das initiale Prozesssystem des I4*Copter*s. Für die Darstellung der Datenflüsse verfügen die Hüllenklassen zudem über *Datenverbinder* (engl. *connector*), welche durch Angabe des Zielprozesses und des zu sendenden Datentyps den einfachen Datenaustausch zwischen Prozessen ermöglichen. Aufgrund der statischen Umsetzung und Ablaufplanung erfolgt die Darstellung der Datenverbinder zur Laufzeit durch *gemeinsamen Speicher* (engl. *shared memory*).

Für die Konfiguration des I4*Copter*-Softwaresystems steht ein umfangreiches Konfigurationssystem zur Verfügung. Dieses erlaubt neben der funktionalen Konfiguration der Anwendung beispielsweise auch die Zuordnung von Anwendungskomponenten zu Prozessinkarnationen des CiAO-Betriebssystems. Ebenso findet hier die Konfiguration der Hardwareschnittstellen und Treiber statt. Die Konfigurationseinstellungen werden dem Erstellungsprozess als Präprozessordirektiven (engl. *preprocessor directives*) zugeführt.

Softwaresystem und Fluglageregelung

Der relevante Teil des Softwaresystem besteht aus drei Anwendungskomponenten respektive Prozessinkarnationen des Betriebssystems. `SignalProcessing` (3 ms) dient der Abtastung und Vorverarbeitung der Messwerte. `FlightControl` (9 ms) umfasst die eigentlichen Regelungsalgorithmen und liefert die Stellwerte für die Antriebsmotoren. `CopterControl` (21 ms) realisiert die Kommunikation mit der Basisstation und liefert die Führungsgrößen des Piloten. Die Datenflüsse zwischen den Komponenten sind durch gemeinsamen Speicher realisiert.

5.2 Replikation der Fluglageregelung

Die Fluglageregelung repräsentiert den sicherheitskritischen Kern des I4*Copter*s und markiert damit auch den Ausgangspunkt für die im Folgenden näher beschriebene exemplarische Anwendung des CoRed-Ansatzes. Hierzu erfolgt zunächst die Bereitstellung der Dienstschicht auf der TriCore-Ausführungsplattform. Damit einher geht die Anpassung der genutzten Treiber und Systemfunktionen sowie die eigentliche Replikation der Prozessinkarnationen der Anwendung. Die erneute Evaluation dient im Anschluss der Überprüfung der erwarteten Eigenschaften des CoRed-Mehrheitsentscheiders. Abschließend wirft dieses Kapitel einen Blick auf die mögliche Erweiterung des Redundanzbereichs über die Grenzen des TriCore-Rechensystems hinaus.

5.2.1 Umsetzung der CoRed-Dienstschicht

Grundsätzlich entspricht die CoRed-Dienstschicht im I4*Copter* der in Kapitel 3.7.1 vorgestellten Basisimplementierung aus Replikationsinfrastruktur, Schlüsselverwaltung und Strukturelementen für die Eingabe und Ausgabe.

Replikation und Komposition

Die CoRed-Dienstschicht gliedert sich in den Konfigurations- und Erstellungsprozess des I4*Copter*s ein und ergänzt diesen durch die bereits erwähnten Möglichkeiten der mehrfachen Instantiierung

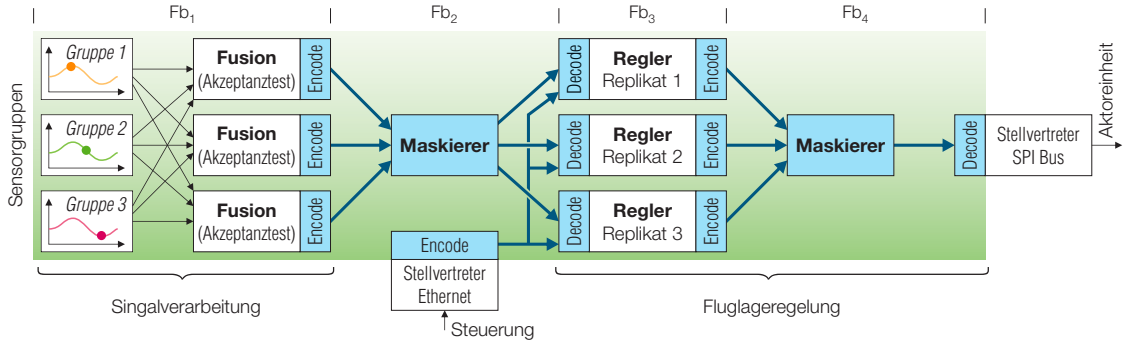


Abbildung 5.3: Schematische Darstellung der replizierten Fluglageregelung des I4*Copter*s.
Für die Replikation wurde die Signalverarbeitung in zwei Schritte aufgeteilt. Die Treiber für die Abfrage der redundanten Sensoren bilden nun drei *Gruppen* (Replikate). Die Sensorfusion dient gleichzeitig als Akzeptanztest für die Messwerte der Sensorgruppen und bildet mit dem nachfolgenden Maskierer einen Akzeptanzentscheider. Die Fluglageregelung erhält neben den konsolidierten Messwerten die Führungsgrößen der Basisstation durch einen einfachen Stellvertreter. Im Anschluss an den finalen Mehrheitsentscheid werden die Stellwerte über einen Stellvertreter an den SPI-Kommunikationsbus ausgegeben.

und Replikation durch C++-Templates. Die Absicherung der Datenflüsse erfolgt durch die geeignete Erweiterung der zwischen den Prozessen genutzten Datenverbinder. Diese erlauben nunmehr den Einsatz codierter Datentypen für die transparente Codierung und Decodierung im Zuge des Datenaustauschs. Der Mehrheitsentscheider wird den Datenverbindern im Bedarfsfall automatisch zwischengeschaltet. Komplexere Zusammenhänge lassen sich darüber hinaus manuell in die Anwendungsbeschreibung und Konfiguration aufnehmen. Die Schlüsselverwaltung weist den codierten Datentypen und Mehrheitsentscheidern im Zuge des Erstellungsprozesses wie bereits beschrieben geeignete Codierungsparameter zu.

Treiber, Systemaufrufe und Kommunikationsdienste

Einen Großteil der Dienstschicht entfällt auf die Emulation der verschiedenen Systemdienste und Hardwaretreiber. Für die im Rahmen der Fallstudie benötigten Teile wurden daher geeignete Eingangsreplikatoren beziehungsweise Stellvertreter für die Ausgänge umgesetzt. Hierzu zählen insbesondere codierte Varianten für die von der Sensorik und Aktorik genutzten Hardwareschnittstellen. Zudem wurden die durch das System angebotenen Kommunikationsdienste geeignet erweitert, um die Nachrichten im Zuge des Empfangs zu codieren beziehungsweise während des Versands zu decodieren.

5.2.2 CoRed-gesicherte Fluglageregelung

Das Ergebnis der Replikation der sicherheitskritischen Fluglageregelung mit CoRed ist schematisch in Abbildung 5.3 dargestellt. Wie erwartet, erweist sich die redundante Auslegung der eigentlichen Fluglageregelung als vergleichsweise unproblematisch. Sowohl die Vervielfältigung der Prozessinkarnationen als auch die Codierung der Datenflüsse erfolgt durch die Replikationsinfrastruktur der CoRed-Dienstschicht. Die Ausgabe der Stellwerte im Anschluss an den finalen Mehrheitsentscheider erfolgt durch den generischen Stellvertreter für den SPI-Kommunikationsbus. Dieser übernimmt die nicht weiter replizierbare Interaktion mit der entsprechenden Peripherieeinheit des TriCores und leitet die Daten letztlich an die Aktoreinheit weiter. In ähnlicher Weise bindet der Stellvertreter für die Ethernet-Schnittstelle die Kommunikation mit der Basisstation an den Redundanzbereich der Fluglageregelung an. Dieser dient als Eingangsreplikator und codiert

die enthaltenen Nutzdaten direkt im Anschluss an den Paketempfang. Die mögliche Eliminierung der beiden Stellvertreter und die damit einhergehende Erweiterung des Redundanzbereiches ist darüber hinaus Gegenstand von Kapitel 5.3.

Die Anbindung der Signalverarbeitung erforderte indes einige Anpassungsmaßnahmen: Deren direkte Replikation erwies sich als nicht zielführend, da das Sensorsystem des I4*Copter*s inhärent redundant ausgelegt ist. Die gemeinsame Abtastung aller Messwerte wurde in der Folge aufgegeben und die Sensoren stattdessen in drei Sensorgruppen eingeteilt. Jede dieser Gruppen repräsentiert nunmehr eine in sich vollständige Inertial-Sensorik, deren Auswertung jeweils in einer eigenen Schutzdomäne erfolgt (vgl. Abbildung 5.3, links). Ziel dieser Maßnahme ist die Vermeidung der Fehlerausbreitung zwischen den logischen Sensorsystemen. Aus Sicht des CoRed-Entwurfs ergibt sich aus dieser Konstellation ein nichtdeterministischer Eingang, welcher entsprechend durch einen Akzeptanzmaskierer behandelt werden muss. Diese Funktion erfüllt konzeptionell die zur Messwertkonsolidierung eingesetzte Datenfusion beziehungsweise die darin enthaltenen Bewertungsfunktionen. Für den Einsatz als Akzeptanztest wurden die entsprechenden Werte durch eine erweiterte Schnittstellendefinition zugänglich gemacht. Die Ausführung der Signalvorverarbeitung und Sensorfusion erfolgt wiederum repliziert. Am Ende der Signalverarbeitung stehen die konsolidierten und codierten Messwerte für die Nutzung in der Fluglageregelung zur Verfügung.

Die beschriebenen Anpassungen der Regelungsanwendung scheinen zunächst im Gegensatz zu der angestrebten Anwendungstransparenz zu stehen. An dieser Stelle ist jedoch zu bedenken, dass die Gruppierung sowie die Akzeptanztests eine generelle Voraussetzung für die Replikation derartiger Eingänge darstellen. Insgesamt veranschaulicht das Fallbeispiel der Fluglageregelung die grundsätzliche Eignung des CoRed-Ansatzes für die Absicherung entsprechend gearteter Regelungsanwendungen.

Fluglageregelung mit CoRed

Die Integration von CoRed in das I4*Copter*-System ermöglicht die einfache Absicherung der kritischen Fluglageregelung. Die Replikation und Codierung der zu schützenden Prozessinkarnationen und Datenflüsse erfolgt dabei zusammen mit der Schlüsselverwaltung im Zuge des Erstellungsprozesses. Die für das Fallbeispiel notwendigen Anpassungen beschränken sich auf die unvermeidbare Behandlung der nichtdeterministischen Sensoreingänge.

5.2.3 Evaluation der CoRed Implementierung

Grundsätzlich wurde die Effektivität der Fehlererkennung sowohl der arithmetischen Codierung als auch des CoRed-Mehrheitsentscheiders bereits in Kapitel 4 gezeigt. Die Ergebnisse der Fehlersimulation gelten hierbei unabhängig von der gewählten Hardware-Plattform beziehungsweise des durch den Compiler erzeugten Binärprogramms. Daher dient die erneute Injektion von Fehlern auf der Befehlssatzebene des TriCore TC1796 in diesem Zusammenhang der Überprüfung der erwarteten Eigenschaften auf der für den I4*Copter* genutzten Ausführungsplattform. Das Vorgehen entspricht dabei weitgehend dem aus Kapitel 4.3.

Fehlerinjektion mithilfe des Diagnose-Programms

Für die Durchführung der Fehlerinjektion auf Befehlssatzebene scheidet das Fail*-Werkzeug im Falle des TriCores aus, da die Unterstützung dieser Hardwareplattform durch den Fail*-Unterbau (engl. *back end*) bislang noch unvollständig umgesetzt und erprobt ist. Um dennoch eine

entsprechende Fehlerinjektionskampagne für die Revalidierung der Erkennungsleistung der arithmetischen Codierung und des CoRed-Mehrheitsentscheiders zu ermöglichen, wurde daher auf die OCDS-Schnittstelle (engl. *on-chip debug support*) [Inf07] des Mikrocontrollers zurückgegriffen. Diese erlaubt die direkte Injektion von Fehlern in den Hardwarekontext des realen Hardwaresystems mithilfe eines *Hardware-Diagnose-Programms* (engl. *debugger*). Im konkreten Fall wurde hierfür das Diagnose-Programm TRACE32 von Lauterbach[3] genutzt. Die Beschreibung der Kampagne erfolgte hierbei durch die werkzeugspezifische Skriptsprache PRACTICE des TRACE32 Diagnose-Werkzeugs. Der Testablauf entspricht ansonsten dem Vorgehen in FAIL* und umfasst die Durchführung eines fehlerfreien Testlaufs, das eigentliche Experiment sowie die anschließende Auswertung des Resultats.

Versuchsaufbau und Reduzierung des Fehlerraums

Der allgemeine Versuchsaufbau gleicht der IA32-Kampagne aus Kapitel 4.3 und basiert auf demselben C++-Quellcode des CoRed-Mehrheitsentscheiders sowie der arithmetischen Codierung. Für die Evaluierung der Einzelbitfehler auf der Befehlssatzebene wurde wiederum der CoRed-Mehrheitsentscheider mit seinem ungeschützten Pendant verglichen. Die Übersetzung des Quellcodes erfolgte dabei durch den GCC Kreuzkompilierer (engl. *cross compiler*) für die TriCore-Plattform in der Version 4.5.2 (Mac OS X). Die Compileroptionen entsprachen dabei mit Ausnahme der Optimierungsstufe (`-O2`, volle Optimierung) den Standardeinstellungen. Die sich ergebenden Größenverhältnisse der jeweiligen Binärprogramme sind in Tabelle 5.1 dargestellt und entsprechen grundsätzlich den Werten der IA32-Plattform aus Kapitel 4.3.2. In absoluten Zahlen benötigt der CoRed-Mehrheitsentscheider auf der TriCore-Plattform hingegen weniger Instruktionen (72 versus 92). Dies begründet sich in der deutlich höheren Zahl an allgemeinen Registern des TriCores, welche sich insbesondere im Umgang mit der Vielzahl von Werten und Codierungsparametern vorteilhaft auswirkt und ein Umherschieben der Daten zwischen Registern und Speicher vermeidet.

Mehrheitsentscheider:	**Einfach**	**CoRed**
Instruktionen	28	72
Binärcode (Größe)	68 Byte	200 Byte

Tabelle 5.1: TriCore-Plattform: Größenvergleich einfacher versus CoRed-Mehrheitsentscheider (32-Bit-Codewörter).

Der zu evaluierende Fehlerraum wird von TRACE32 im Zuge des Testablaufs schrittweise durch eine vollständige Iteration über die möglichen Injektionspunkte aufgespannt – eine global angelegte Reduzierung des Fehlerraums ist damit jedoch nicht möglich. Um die Zahl der Experimente dennoch in handhabbaren Grenzen zu halten, wurde eine manuelle Einschränkung des Fehlerraums durchgeführt. Hierbei wurden analog zu dem Vorgehen von Rebaudengo und Reorda [RR99] ungenutzte Register und Zeiträume aus der Testfallgenerierung ausgeschlossen. Diese Methode ist jedoch weit weniger effektiv als die systematische Reduzierung in FAIL*, weshalb statt der 30 000 Experimente für die IA32-Kampagne nunmehr über 230 000 Experimente für die TriCore-Plattform notwendig waren. Nachteilig bei der Fehlerinjektion über die Diagnoseschnittstelle des Mikrocontrollers ist zudem der mit annähernd einer Sekunde pro Einzelexperiment enorme Zeitaufwand und die fehlende Möglichkeit zur Parallelisierung der Experimente.

[3] Hersteller Webseite: `www.lauterbach.de`, Abruf: 18.02.2014

		Einfacher Mehrheitsentscheider		CoRed-Mehrheitsentscheider	
		Datenregister	Adressregister	Datenregister	Adressregister
Ohne Auswirkung (verdeckter Defekt)		1583	882	859	369
Erkannte Fehler	CoRed (arithmetische Codierung)	–	–	7056	1556
	Schutzverletzung / Hardware Ausnahme	832	6413	1338	14 732
	Terminüberschreitung	0	0	0	0
Unerkannte Datenfehler		**634** ↯	**589** ↯	0	0
	$\sum$	3049	7884	9253	16 657

Tabelle 5.2: Fehlerinjektion von Einzelbitfehlern – Evaluation des CoRed-Mehrheitsentscheiders im I4*Copter*.
Überprüfung der erwarteten Eigenschaften des CoRed-Mehrheitsentscheiders ($A = 58\,659$) durch Fehlerinjektion auf der im I4*Copter* genutzten TriCore-Plattform. Die Kampagne deckt dabei den gesamten Fehlerraum für Einzelbitfehler ab. Durch die Harvard-Architektur des TriCore untergliedern sich die Ergebnisse der Fehlerinjektion in Daten- und Adressregister. Insgesamt bestätigt sich auch auf der TriCore-Plattform das erwartete Verhalten und die Robustheit des CoRed-Mehrheitsentscheiders.

Ergebnisse der Fehlerinjektion für den CoRed-Mehrheitsentscheider

Für die Überprüfung der Effektivität des CoRed-Mehrheitsentscheiders auf dem TriCore-Rechensystem wurde wiederum ein Vergleich der beiden Varianten des Entscheiders durchgeführt. Die Fehlerinjektion umfasste hierbei den gesamten Fehlerraum für Einzelbitfehler. Die Messergebnisse für die effektiven Fehler der Kampagne sind in Tabelle 5.2 dargestellt, wobei sich die Fehlerfolgen in die bereits in Kapitel 4.3.2 eingeführten Kategorien für verdeckte, erkannte und unerkannte Fehler gliedern. Aufgrund der Größenunterschiede zwischen den beiden Varianten des Mehrheitsentscheiders weicht die Zahl der Experimente voneinander ab. Da die TriCore-Prozessorfamilie eine *Harvard-Architektur* mit getrennten Daten- und Adressbussen aufweist, unterteilen sich die Ergebnisse der Fehlerinjektion nunmehr zusätzlich in Daten- und Adressregister.

Wie schon bei den Experimenten für die IA32-Plattform fällt zunächst die große Zahl der verdeckten Fehler auf, welche sich aus der inhärenten Redundanz der Mehrheitsfindung ergibt. Im Vergleich liegt die Zahl der effektiven Fehler trotz der geringeren Größe des Binärprogramms deutlich höher. Der direkte Vergleich der Messergebnisse ist jedoch aufgrund der unterschiedlichen Fehlerinjektionswerkzeuge und Hardwareplattformen problematisch. Eine mögliche Erklärung ist jedoch die größere Zahl von 32 allgemeinen Registern des TriCores. Diese erlaubt dem Compiler die exklusive Nutzung von Registern für einen Großteil der Werte und Codierungsparameter, was letztlich die Angriffsfläche vergrößert und damit potenziell auch die Wahrscheinlichkeit eines effektiven Fehlers steigert. Unabhängig davon bestätigt sich die Anfälligkeit des einfachen Mehrheitsentscheiders gegenüber unerkannten Datenfehlern. Deren Anteil beläuft sich im Mittel auf über elf Prozent der effektiven Fehler. Wie erwartet, ist die arithmetische Codierung in der Lage alle verbleibenden Fehler zuverlässig aufzudecken. Entsprechend treten auch auf der TriCore-Plattform *keine unerkannten Datenfehler* innerhalb des CoRed-Mehrheitsentscheiders auf.

Die durch die Harvard-Architektur des TriCores bedingte Trennung in Adress- und Datenregister erlaubt darüber hinaus einen Einblick in den Wirkungsbereich der jeweiligen Schutzmaßnahmen. Die hardwarebasierte Fehlererkennung und Isolation deckt konzeptbedingt in erster Linie adressbezogene Fehler in Form von ungültigen Zugriffen auf. Datenfehler werden für die Speicherschutzeinheit des TriCore beispielsweise erst erkennbar, wenn diese für Adressberechnungen genutzt werden und damit mittelbar zu Schutzverletzungen führen. Demgegenüber erkennt die arithmetische Codierung eine große Zahl an Fehlern in Datenregistern.

Zusammenfassend verhält sich die Umsetzung von CoRed auch nach dem Wechsel der Ausführungsplattform entwurfsgemäß. Die Ergebnisse der Fehlerinjektion geben damit keine Hinweise auf für die Robustheit relevanten Unterschiede zwischen den Architekturen.

Effektive Fehlererkennung – Auch auf der TriCore-Plattform

Die Fehlerinjektion erfolgt im Falle des TriCores mithilfe des TRACE32 Diagnose-Programms direkt auf der Hardware. Die Ergebnisse sind dabei konsistent mit denen der IA32-Kampagne und zeigen die erwartete *zuverlässige* und *vollständige* Fehlererkennung aller Einzelbitfehler. Der Wechsel der Ausführungsplattform erweist sich insgesamt als unproblematisch.

5.2.4 Kostenbetrachtung

Die folgende Betrachtung der durch CoRed induzierten Kosten geht von der Annahme aus, dass eine vergleichbare reguläre Replikation der Prozessinkarnationen ohne CoRed stets unter den selben Voraussetzungen erfolgt. Dies umfasst insbesondere die Beibehaltung der Fehlerbereiche und die damit verbundene räumliche und temporale Isolation durch das Betriebssystem. Bei einer funktionalen Äquivalenz ergeben sich die Kosten im Sinne zusätzlicher Berechnungen beziehungsweise einem erhöhten Speicherverbrauch letztlich alleine aus der Codierung der Datenflüsse und den eingesetzten CoRed-Mehrheitsentscheidern.

Operation:	`encode()`	`decode()`	`check()`	`cored_vote()`
Instruktionen	4	26	16	72
Binärcode (Größe)	14 Byte	86 Byte	52 Byte	200 Byte
Laufzeit (ungünstigst)	0,0261 µs	0,2324 µs	0,1215 µs	0,3799 µs

Tabelle 5.3: Größe und Laufzeitverhalten der CoRed-Basisoperationen (32-Bit-Codewörter).

Die Codegrößen und Laufzeiten der CoRed-Grundbausteine sind in Tabelle 5.3 dargestellt. Wie schon bei den Messungen für den CoRed-Mehrheitsentscheider handelt es sich hierbei um statisch parametrierte Funktionen, wie sie durch die CoRed-Schlüsselverwaltung und den Compiler im Zuge des Erstellungsprozesses erzeugt werden. Allgemein ist der Speicherbedarf der Codewörter zu berücksichtigen, welcher je nach der gewählten Aufteilung zwischen Nutz- und Prüfinformation üblicherweise eine Verdopplung der Wortbreite mit sich bringt. Trotz der im Verhältnis zum einfachen Mehrheitsentscheider 2,6-fachen Codegröße des CoRed-Mehrheitsentscheiders (vgl. Tabelle 5.1) erweisen sich die absoluten Werte der Basisfunktionen als klein beziehungsweise kurz im Sinne der Ausführungszeit.

Die tatsächlichen Gesamtkosten sind des Weiteren in hohem Maße von der Struktur der Anwendung abhängig. Um die Zahlen in einen entsprechenden Kontext zu setzen, wurden daher exemplarisch die Laufzeitkosten der CoRed-Basisoperationen sowie der replizierten `FlightControl`-Komponente betrachtet. Die folgenden Messungen wurden hierbei experimentell mithilfe des Systemzeitgebers (engl. *system timer*) des TriCores ermittelt. Die Werte stellen jeweils das Maximum über 100 Versuche dar.

Die Replikate der eigentlichen Fluglageregelung umfassen in der Summe 18 Eingänge und drei Ausgänge sowie den finalen Mehrheitsentscheider. Im Vergleich mit der einfachen Replikation erhöhen sich die Laufzeitkosten durch CoRed in diesem Beispiel um näherungsweise fünf Mikrosekunden. Im Vergleich dazu summiert sich die Ausführung des Regelalgorithmus (je

187,4 μs) sowie die notwendigen Kontextwechsel durch das CiAO-Betriebssystem (je 1,9 μs) auf über 571 Mikrosekunden. In diesem realistischen Szenario liegen die durch CoRed verursachten Kosten dementsprechend bei *unter einem Prozent* und erscheinen daher auch im Kontext ressourcenbeschränkter, eingebetteter Systeme als grundsätzlich handhabbar.

Duplexbetrieb mit Bereitschaftsreplikat

Aufgrund der vergleichsweise geringen Fehlerraten für transiente Fehler, stellt die Fehlerbehandlung und -toleranz in der Regel den seltenen Ausnahmefall dar. Für die in Kapitel 3.2 getroffene Einzelfehlerannahme genügt der Einsatz von zwei Replikaten und der Vergleich auf Übereinstimmung der Ausgaben für die entscheidende Fehlererkennung. Erst im Fehlerfall benötigt der Relativtest des Mehrheitsentscheiders das Ergebnis des dritten Replikats für die Entscheidungsfindung und Fehlerdiagnose. Durch einen entsprechenden *Duplexbetrieb mit einem Bereitschaftsreplikat* (engl. *duplex with stand-by*) [Ech90] lassen sich somit im Normalbetrieb rund ein Drittel der Kosten einsparen.

Die Anwendung dieses Verfahrens erfordert jedoch die anwendungsspezifische Erstellung eines geeigneten Rücksetzpunkts in Form eines korrekten und aktuellen Replikatzustands. Dieser wird im Fehlerfall für die Aktualisierung des Bereitschaftsreplikats genutzt, um anschließend die Berechnung mit den gegenwärtigen Eingaben fortzusetzen. Die Herausforderung liegt hierbei in der geeigneten Bewertung und Überprüfung des Replikatzustands, da sich bösartige Defekte und Fehler nicht zwangsläufig bis an die Außengrenzen des betroffenen Replikats ausbreiten müssen. Sie sind somit für den regulären Mehrheitsentscheid nicht erkennbar. Im konkreten Fall der Fluglageregelung des I4*Copter*s erlaubt eine geeignete Konfiguration des Quellcode-Generators von MATLAB/SIMULINK ein direktes Auslesen und Setzen des Anwendungszustands durch entsprechende Zugriffsfunktionen. Der notwendige Zustandsvergleich lässt sich dabei auf das bekannte CoRed-Schema abbilden: Die Replikate werden um einen weiteren Ausgang erweitert, welcher den jeweiligen Zustand in codierter Form ausgibt. Dieser wird bei Übereinstimmung gesichert und dient im Fehlerfall als codierter Eingabeparameter für das Bereitschaftsreplikat. Diese Art der Umsetzung erhält somit die Fehlererkennung durch CoRed auch für den notwendigen Zustandstransfer zwischen Normalbetrieb und Fehlerbehandlung.

5.3 Erweiterung des Redundanzbereichs über Systemgrenzen

Die Absicherung der Fluglageregelung des I4*Copter*s zeigt anschaulich die Anwendungsmöglichkeiten von CoRed innerhalb eines Rechensystems auf. Die bislang erforderlichen Eingänge beziehungsweise Ausgänge des Redundanzbereichs beschränken sich jedoch auf den Einsatz von Replikatoren und Stellvertretern. Um darüber hinaus auch die in Kapitel 3.4 beschriebenen *codierten Eingänge* beziehungsweise *Ausgänge* in das Fallbeispiel aufnehmen zu können, wurden die im Folgenden näher beschriebenen, zusätzlichen Erweiterungen des I4*Copter*s vorgenommen.

5.3.1 Codierte Ausgabe der Stellwerte

Einen ersten Hinweis auf die mögliche Erweiterung von CoRed über die Grenzen eines Systems hinaus gab bereits die in Kapitel 5.1.1 vorgestellte spezifische Auslegung der Motorplatine als intelligentes Aktorsystem. Dieses übernimmt stellvertretend die Steuerung der Leistungselektronik und erhält die hierfür notwendigen Stellwerte mittels eines SPI-Kommunikationsbusses von der CoRed-geschützten Fluglageregelung. Für die Ausgabe am Ende der replizierten Ausführung bedeutet dies bislang die Übergabe des codierten Gewinners an den Stellvertreter für die SPI-

Buskommunikation. Dieser decodiert den Stellwert für den Nachrichtenversand und legt ihn an einer festgelegten Stelle im Speicher ab. Die eigentliche Übertragung erfolgt anschließend durch die entsprechende Hardwareeinheit des TriCores. Bis zur eigentlichen Verwendung der Stellwerte vergehen jedoch im Mittel 1,4 Millisekunden für die Übertragung der Nachricht und deren Auswertung auf dem Aktorsystem. Selbst bei einer adäquaten Fehlererkennung des Bussystems bleiben transiente Fehler auf dem Weg von und zu dem Transportmedium auf beiden Seiten potenziell unentdeckt – sie stellen somit von Neuem kritische Fehlerstellen dar.

Fehlererkennung beim Aktor

Eine einfache Lösung dieses Problems ist der Einsatz eines codierten Ausgangs und somit die direkte Übertragung des Gewinners zwischen der zentralen Fluglageregelung und dem Aktorsystem. Hierfür wurde die nachgelagerte Steuerung der Frequenzumrichter entsprechend angepasst. Die Decodierung des Gewinners erfolgt nun erst unmittelbar vor der Nutzung durch das Cortex-M3-Rechensystem. Hierfür genügt letztlich der Austausch der notwendigen Codierungsparameter, was im konkreten Fall durch eine Integration des Aktorsystems in den Erstellungsprozess des I4*Copter*s erfolgt. Die CoRed-Schlüsselverwaltung wählt hierbei in einem ersten Durchlauf die Codierungsparameter des finalen Mehrheitsentscheiders aus und übergibt die entsprechenden Werte anschließend an die Decodierungsoperation des Aktorsystems. Durch diese einfache Erweiterung lässt sich der Redundanzbereich von CoRed über die Grenzen des TriCore-Rechensystems erweitern. Entsprechend stellen die jeweiligen Systemdienste für die Kommunikation keine kritischen Fehlerstellen mehr dar. Unter der Annahme, dass der Nachrichtenversand strikt periodisch erfolgt, kann das Aktorsystem somit die Gültigkeit der Stellwerte und die fehlerfreie Funktion der Fluglageregelung eigenständig und zuverlässig ermitteln.

Fehlertolerantes Verhalten

Grundsätzlich ermöglicht die Erweiterung des Redundanzbereichs durch Beibehaltung der arithmetischen Codierung der Datenflüsse die bereits in Kapitel 4.2 nachgewiesene, zuverlässige Erkennung von Fehlern. Ein darauf aufbauendes fehlertolerantes Verhalten erfordert wiederum die Rückwärtskorrektur bis zur letzten Überprüfung der Codierung. Die entsprechende Umsetzung ist hierbei fallspezifisch und hängt in hohem Maß von den technischen Gegebenheiten und der Kommunikationsinfrastruktur ab. Im konkreten Fall überprüft der durch die CoRed-Dienstschicht bereitgestellte codierte SPI-Nachrichtenversand die Korrektheit des Gewinners direkt im Anschluss an den finalen Mehrheitsentscheid. Im Fehlerfall lässt sich dieser somit direkt wiederholen. Die Kommunikation selbst stellt nun ihrerseits einen eigenständigen Fehlerbereich dar. Fehler innerhalb der SPI-Treiber oder der eigentlichen Übertragung lassen sich durch eine Verfälschung des Codeworts oder ein Ausbleiben desselben zuverlässig erkennen und im Fehlerfall durch eine wiederholte Übertragung beheben. Neben der regulären Überwachung durch das Betriebssystem nutzt CoRed hierfür einen Funktionswächter. Dieser wird entweder durch die Fehlerüberprüfung der TriCore-Kommunikationseinheit oder durch eine entsprechende Signalisierung des Aktorsystems ausgelöst und sieht in der Folge die erneute Durchführung der Nachrichtenübertragung vor.

5.3.2 Codierte Eingabe von Messwerten und Steuernachrichten

Analog zu der Codierung der Ausgänge lassen sich auch die Eingänge des Rechensystems durch die arithmetische Codierung der Eingaben erweitern. Zu diesem Zweck wurde der I4*Copter* um eine *codierte Messquelle* erweitert. Darüber hinaus wurde auch die Kommunikation mit der Basisstation und damit die Übertragung der Steuereingaben des Piloten in eine *codierte Steuerung* überführt.

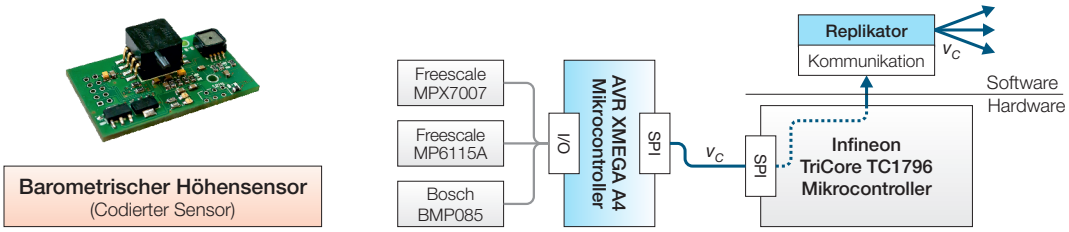


Abbildung 5.4: Versuchsmuster einer codierten Eingabequelle in Form eines barometrischen Höhensensors.
Durch die direkte Codierung der Messwerte auf dem Sensor lassen sich die in Kapitel 3.4.1 eingeführten codierten Eingänge nutzen. Hierdurch entfällt die Notwendigkeit der komplexen Eingangsreplikation auf dem zentralen Steuersystem des I4*Copter*s vollständig. Als Versuchsmuster für einen solchen intelligenten Sensorknoten wurde die barometrische Höhenmessung des I4*Copter*s um ein kleines Rechensystem ergänzt, welches die Messwerte der angeschlossenen Sensoren erfasst und codiert.

Versuchsmuster eines codierenden Sensors

Der in Abbildung 5.4 dargestellte *intelligente Sensorknoten* dient als Versuchsmuster für einen Sensor mit codierter Ausgabe und ersetzt die für die Flughöhenregelung notwendige barometrische Höhenmessung. Für die Auswertung der Drucksensoren dient in diesem Beispiel ein 16-Bit *Atmel AVR XMEGA A4 Mikrocontroller (AVR)* [Atm09] als Basis. Dieser bestimmt die relative Höhe und codiert die Messwerte vor dem Versand an das TriCore-Rechensystem. Für die Kommunikation kommt wiederum der SPI-Kommunikationsbus zum Einsatz. Die für die arithmetische Codierung notwendigen Parameter werden, wie schon bei der codierten Ausgabe der Stellwerte, im Zuge des Erstellungsprozesses ausgetauscht.

Die Codierung der Eingaben erweitert nicht nur den Redundanzbereich auf den Sensorkonten, sondern vereinfacht die Eingangsreplikation auf der Seite der Fluglageregelung signifikant. Der bislang für die Überprüfung und Konsolidierung der Messwerte notwendige Mehrheitsentscheider und dessen Prozessinkarnation entfallen vollständig. Darüber hinaus wird der mehrfach abfragende Replikator durch einen einfachen Stellvertreter für die Buskommunikation ersetzt, wodurch die Replikate nunmehr direkt mit den codierten Messwerten arbeiten. Insgesamt basiert die Fehlererkennung alleine auf der arithmetischen Codierung der Datenflüsse. Ein Versagen der Kommunikation oder eine Verfälschung der Messwerte auf dem Weg vom Sensor zu den Replikaten lässt sich im Zuge der Decodierung aufdecken und entsprechend geeignet behandeln – beispielsweise durch eine erneute Abfrage des Sensors durch den Stellvertreter.

Erweiterung des Redundanzbereichs

Das Versuchsmuster eines codierenden Sensors sowie die Übertragung codierter Stellwerte an das Aktorsystem zeigen anschaulich die Möglichkeiten der Erweiterung des Redundanzbereichs über *Systemgrenzen* hinaus. Durch *codierte Ausgänge* wird eine nachgelagerte Überprüfung der Mehrheitsentscheidung ermöglicht. Umgekehrt eliminieren *codierte Eingänge* einen Großteil der durch die Eingangsreplikation hervorgerufenen Komplexität.

Kommunikation mit der Basisstation

In gleicher Weise lassen sich auch die durch den Piloten vorgegebenen Sollwerte für die Fluglage mit arithmetischer Codierung versehen. Die notwendigen Steuerungsinformationen werden von

der Basisstation mittels WLAN an den I4*Copter* kommuniziert. Die eingehenden Netzwerkpakete durchlaufen dabei die in Abbildung 5.2 (Mitte) dargestellte Kette aus WLAN-Brücke, Netzwerkschnittstelle und externem Speicher des TriCores, bis sie schließlich durch den entsprechenden Netzwerktreiber der Anwendung zugänglich gemacht werden. Die Fehlererkennung der arithmetischen Codierung bezieht sich wiederum ausschließlich auf den Datenfluss und ist in diesem Sinne konzeptionell geeignet die für die Kommunikation notwendigen Verarbeitungsschritte vollständig zu umfassen. Dieses Vorgehen besitzt im Gegensatz zu einer anderweitigen Codierung der Nutzdaten den Vorteil der nahtlosen Integration. Darüber hinaus lässt sich bei einem Mehrheitsentscheid am Sender auch dessen korrekte Funktion nachprüfen.

Unabhängig von dem eigentlichen Gefährdungspotential zeigt die Kommunikation der Steuerungsinformationen jedoch eine grundsätzliche Einschränkung der Datencodierung auf: Im Unterschied zu der vergleichsweise einfachen und zustandslosen SPI-Kommunikation der vorangegangenen Beispiele, ist eine Fehlerdiagnose und damit eine geeignete Behandlung durch das implizite Zusammenfassen mehrerer Teilsysteme im Allgemeinen nicht möglich. Zur Lösung dieses Problems bleibt letztlich nur die redundante Auslegung respektive die stufenweise Absicherung der entsprechenden Kommunikations- beziehungsweise Verarbeitungspfade bis zur eigentlichen Quelle der Eingangsdaten. Gegenüber der ungeschützten Kommunikation verhält sich die codierte Variante aus Sicht der Fluglageregelung jedoch zumindest inaktiv im Fehlerfall (engl. *fail silent*). Dies erlaubt die für einen Abriss der Kommunikation ohnehin vorgesehene Einnahme eines sicheren Zustands in Form einer kontrollierten Landung des Fahrzeugs.

5.4 Zusammenfassung

Regelungssysteme stellen die natürliche Zieldomäne für CoRed dar. Zum einen repräsentieren sie rein quantitativ den überwiegenden Teil der sicherheitskritischen Anwendungen mit einem ausgewiesenen Schutzbedürfnis gegenüber transienten Fehlern. Zum anderen eignen sich Regelungssysteme aufgrund ihres geradlinigen Aufbaus aus Einlesen, Verarbeiten und Ausgeben besonders für die Replikation. Die Fluglageregelung des unbemannten Luftfahrzeugs I4*Copter* ist in diesem Zusammenhang ein forderndes und praxisnahes Fallbeispiel für den Einsatz von CoRed. Das hardwaretechnische Herz stellt dabei der in der Fahrzeugelektronik weitverbreitete Infineon TriCore TC1796 Mikrocontroller dar. Daran angeschlossen, findet sich eine umfangreiche und redundant ausgelegte Inertial-Sensorik sowie ein intelligentes Aktorsystem.

Die Anwendung von CoRed auf die Regelungsanwendung des I4*Copter*s geht entsprechend einher mit einem Wechsel der Ausführungsplattform. Die hierfür angezeigte erneute Überprüfung der Effektivität des CoRed-Mehrheitsentscheiders auf der Befehlssatzebene zeigt, dass die *zuverlässige und vollständige Fehlererkennung* auch auf der TriCore-Plattform gegeben ist. Im Gegensatz zu den Untersuchungen auf der IA32-Plattform wurde die Fehlerinjektion im Falle des TriCores mithilfe eines Diagnose-Programms auf der realen Hardware durchgeführt, was die Ergebnisse der bisherigen Untersuchungen zusätzlich stützt und unterstreicht.

Den eigentlichen Kern des Fallbeispiels stellt jedoch die Replikation der Fluglageregelung dar. Die praktische Anwendbarkeit von CoRed erweist sich dabei im Allgemeinen als sehr gut. Die redundante Auslegung der eigentlichen Regelung erfordert lediglich die Vervielfältigung der entsprechenden Prozessinkarnation und den nachgeschalteten Mehrheitsentscheid. Die Fallstricke liegen in diesem Beispiel in den Eingängen in Form der Sensorik und Signalverarbeitung. Hier erweist sich ausgerechnet das gezielt redundant ausgelegte Sensorsystem als hinderlich für eine transparente Replikation. Die Gründe liegen vor allem in dem nichtdeterministischen Verhalten der Messwerte untereinander. Dieses erfordert den Einsatz eines Akzeptanzmaskierers an den Eingängen und führt somit zum einen zu einer Auftrennung von Messwertabfrage und Signalver-

arbeitung. Zum anderen setzt dies einen geeigneten Akzeptanztest voraus, welcher im konkreten Fall zunächst nicht zugänglich war – obwohl er durch die Sensorfusion grundsätzlich vorhanden ist. Die entsprechende Anpassung der Signalverarbeitung stellt jedoch eine allgemeine Voraussetzung für deren Replikation dar und ist nicht spezifisch für CoRed.

Die durch CoRed induzierten Kosten sind vergleichsweise klein. Für die replizierte Fluglageregelung (nur `FlightControl`) ergeben sich zusätzliche *Laufzeitkosten von unter einem Prozent* im Vergleich zu einem funktionsäquivalenten, regulären TMR-System. Darüber hinaus bietet der *Duplexbetrieb mit Bereitschaftsreplikat* ein großes Einsparungspotential. Die Replikation ist hierbei optimistisch und führt das dritte Replikat lediglich im Fehlerfall aus. Der notwendige Zustandsvergleich lässt sich dabei in idealer Weise auf das reguläre CoRed-Schema abbilden: Die Zustände des Duplexsystems werden zu Ausgängen und somit zu einem regulären Teil des Mehrheitsentscheids. Der fehlerfreie und durch die Codierung geschützte Zustand des vorangegangenen Durchlaufs dient im Fehlerfall als Eingang für das Bereitschaftsreplikat.

Die grundsätzliche Erweiterbarkeit des durch CoRed aufgespannten Redundanzbereichs zeigen anschaulich das Versuchsmuster eines codierenden Sensors sowie die Ausgabe codierter Stellwerte an das Aktorsystem auf. Letzteres ermöglicht sowohl die Überprüfung der Daten als auch des Mehrheitsentscheids bis kurz vor der Verwendung der Stellwerte direkt auf dem Aktor. Umgekehrt erübrigt sich durch den Einsatz codierter Eingänge ein großer Teil der aufwendigen Eingangsreplikation, wie zum Beispiel der für das I4*Copter*-Sensorsystem notwendige Akzeptanzmaskierer. Die Realisierung entsprechender Quellen beziehungsweise Senken erweist sich hierbei als vergleichsweise einfach. Der Charme liegt dabei weniger in der arithmetischen Codierung der Daten an sich, sondern vielmehr in den nahtlosen Übergängen zwischen Datenübertagung, Berechnung und Mehrheitsentscheidung.

6

Weiterführende Betrachtung der Replikation von Regelungsanwendungen

Combined Redundancy for Control (CoRed@Control)

Die Anwendung des CoRed-Ansatzes auf die Fluglageregelung des I4*Copter*s im vorangegangenen Kapitel zeigte anschaulich die Machbarkeit und Effektivität der softwarebasierten Fehlertoleranz in einem praxisnahen Anwendungsfall. Von dieser Betrachtung ausgenommen war bislang die Frage nach einer möglichen gegenseitigen Beeinflussung von Fehlertoleranz und Regelungsanwendung. Deren naive Replikation erweist sich als überaus problematisch, da die Verbesserung der *Zuverlässigkeit* durch den vergrößerten Laufzeitbedarf der redundanten Ausführung in direkter Wechselwirkung mit der nicht-funktionalen Eigenschaft *Rechtzeitigkeit* steht. Abbildung 6.1 veranschaulicht den resultierenden negativen Effekt auf die Regelungsanwendung am Beispiel der Fluglageregelung des I4*Copter*s in Kapitel 5. Die Güte der Regelung nimmt im replizierten Fall signifikant ab und es kommt zu einem unerwünschten Regelverhalten in Form von Schwingungen um den Sollwert. Hervorgerufen wird diese Abweichung durch den im Vergleich zu der einfachen Ausführung deutlich größeren zeitlichen Abstand zwischen der Parametererfassung und der Stellwertausgabe [SC95]. Gängige Entwurfsverfahren stimmen die Regelungsanwendung auf ein bestimmtes zeitliches Verhalten ab [Tör98, ÅW11]. Ihre Vervielfachung führt daher zu einer Verletzung der zeitlichen Annahmen und damit potenziell zu der in Abbildung 6.1 exemplarisch dargestellten Herabsetzung der Leistungsfähigkeit der Regelungsanwendung.

Die redundante Auslegung der Regelung stellt dabei kein grundsätzliches Entwurfsproblem dar und ist eine gängige Maßnahme in sicherheitskritischen Systemen [Yeh96]. Die Redundanz wird hier üblicherweise direkt auf der Anwendungsebene [ABK+93, BG97, BKL+10] eingefügt, wodurch diese einen integralen Bestandteil des Reglerentwurfs darstellt. Somit ist eine direkte Berücksichtigung des zeitlichen Verhaltens der redundanten Regelung möglich. Im Gegensatz dazu führt CoRed die Replikation *nachträglich* in Form zusätzlicher Prozessinkarnationen und ohne Kenntnis der Anwendungseigenschaften auf der Ebene des Betriebssystems durch. Es ist somit keine Rückkopplung zwischen der eingefügten Redundanz sowie der hierdurch veränderten

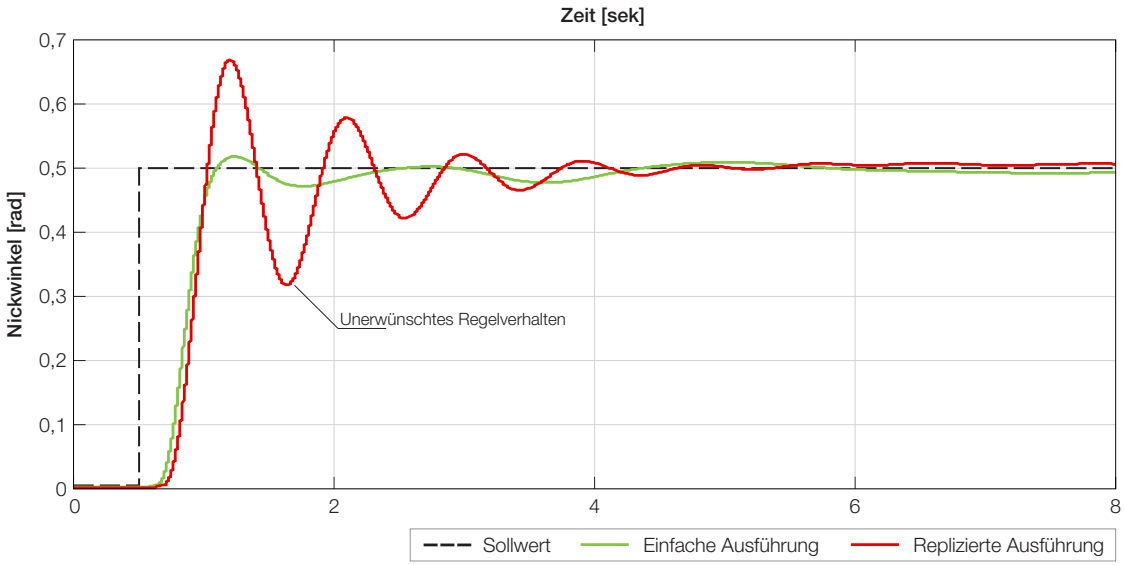


Abbildung 6.1: Auswirkung der replizierten Ausführung auf die Regelgüte am Beispiel einer Fluglageregelung.
Durch den höheren Laufzeitbedarf der replizierten Ausführung entsteht ein durch die Regelungsanwendung nicht berücksichtigter zeitlicher Abstand zwischen Parametererfassung und Stellwertausgabe. In der Folge kann es zu einer signifikanten Herabsetzung der Regelgüte kommen. Im Falle der dargestellten Fluglageregelung zeigt sich folglich ein im Vergleich mit der unreplizierten Variante ausgeprägtes Schwingungsverhalten bei der Anpassung der Führungsgröße.
Ergebnis einer Simulation von Modell und Regelstrecke in MATLAB/SIMULINK *des I4Copter-Fluglagereglers. Der zeitliche Versatz der beiden Varianten wurde aus Messungen auf der Hardware abgeleitet beziehungsweise abgeschätzt.*

zeitlichen Eigenschaften und den Modellannahmen der Regelungsanwendung gegeben. Aufgrund der Wechselbeziehungen zwischen der Zuverlässigkeit und den zeitlichen Eigenschaften der Anwendung ist die angestrebte Transparenz im Falle der Regelungsanwendungen nicht notwendigerweise gegeben. Zur Lösung dieses Problems bleibt letztlich nur die Minimierung der Abweichung zwischen den zeitlichen Annahmen der Regelungsanwendung und den tatsächlichen Ausführungsbedingungen. Ein möglicher Weg ist hierbei die starke Überdimensionierung des Rechensystems, um den zusätzlichen Rechenzeitbedarf auf ein erträgliches Niveau zu senken. Der im Rahmen dieser Arbeit verfolgte Lösungsansatz zielt jedoch auf die Bereitstellung einer *geeigneten Schnittstelle* zwischen der Regelungsanwendung und der Ausführungsumgebung und somit den Austausch der notwendigen Eigenschaften ab.

Das Ziel dieses Kapitels ist daher eine weitergehende Betrachtung der Replikation von Regelungssystemen und der sich hieraus ergebenden Probleme. Zu diesem Zweck erfolgt zunächst eine erweiterte Problemanalyse zur Charakterisierung der Modellannahmen und Abweichungen sowie der generellen Struktur von Regelungsanwendungen in Kapitel 6.1. Darauf aufbauend zeigt Kapitel 6.2 die Erweiterung des CORED-Ansatzes zu CORED@CONTROL (engl. *Combined Redundancy for Control Applications*) und die damit einhergehende Einführung einer qualitätsgewahren Anwendungsschnittstelle zwischen der Regelungsanwendung und der Ausführungsumgebung. Darauf aufbauend entwickelt Kapitel 6.3 eine auf die Anforderungen der Replikation angepasste Anwendungsarchitektur. Das Ziel hierbei ist eine geeignete Aufteilung und Abbildung der Regelungsanwendung auf replizierbare Prozessinkarnationen zu unterstützen und darüber hinaus die zeitlichen Abweichungen an geeigneter Stelle zurückzuführen. Die feingranulare Modularisierung der Regelungsanwendung dient in diesem Zusammenhang auch der Senkung der Fehlererkennungslatenzen.

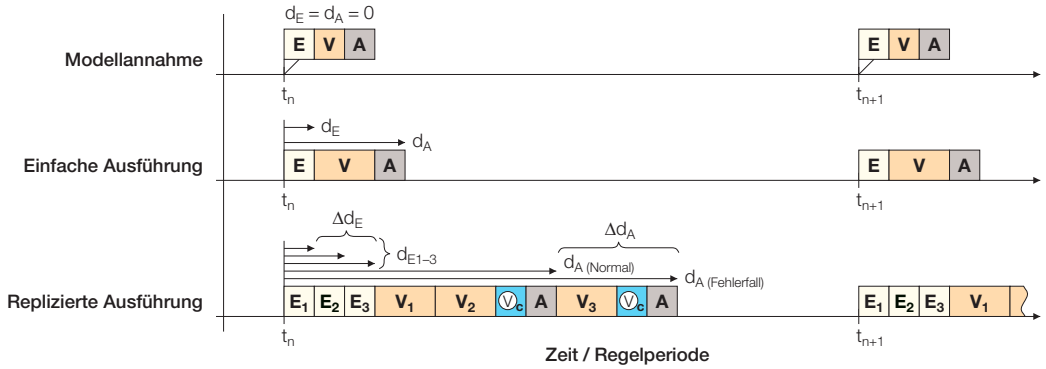


Abbildung 6.2: Diskrepanz zwischen regelungstechnischen Modellannahmen und tatsächlicher Ausführung.
Das Lesen der Sensoren (Eingabe, E), die Berechnung des Stellwerts (Verarbeitung, V) und das Stellen des Aktors (Ausgabe, A) erfolgen bei gängigen regelungstechnischen Entwurfsverfahren periodisch und werden als gleichzeitig und verzögerungsfrei angenommen (oben). Diese vereinfachenden Modellannahmen lassen sich in einem Einfachsystem näherungsweise durch eine entsprechende Auslegung und Ablaufplanung erreichen (Mitte). Bei der replizierten Ausführung steigt der Laufzeitbedarf und damit die Zeitspanne bis zur Ausgabe (d_A) signifikant an. Darüber hinaus ergeben sich Abweichungen in Abtastzeitpunkten der Sensoren (Δd_E) und bei einer optimistischen Ausführung auch bei der Stellwertausgabe im Fehlerfall (Δd_A).

6.1 Erweiterte Problemanalyse

Regelungsanwendungen zeichnen sich durch ihre Kopplung an ein zu kontrollierendes physikalisches System aus: Sie nutzen Sensoren für die Erfassung des aktuellen Zustands der Umwelt, verarbeiten deren Daten und nutzen Aktoren zur gezielten Beeinflussung des zu kontrollierenden physikalischen Prozesses. Neben einer zuverlässigen Ausführung durch das Rechensystem sind sie daher auch maßgeblich von der rechtzeitigen Bereitstellung der Ergebnisse abhängig.

6.1.1 Diskrepanz zwischen Modellannahmen und Ausführung

Eine übliche, stark vereinfachende Grundannahme beim Entwurf von Regelungssystemen ist in diesem Zusammenhang die äquidistante und gleichzeitige Ausführung aller Elemente der Regelung wie Sensorik, Stellwertberechnung oder Aktorik [FPW97, Tör98, KG02, ÅW11], wie sie in Abbildung 6.2 (oben) in Form eines zusammenhängenden Eingabe-Verarbeitung-Ausgabe-Blocks dargestellt ist. Eine ähnliche Modellannahme findet sich in der Informatik unter dem Begriff der synchronen Hypothese [PBdST05]. Diese Annahme ist jedoch schon aufgrund der unvermeidbaren Ausführungszeiten nicht exakt umsetzbar und führt zu Abweichungen zwischen den regelungstechnischen Annahmen und den tatsächlichen Eigenschaften der Ausführung in Form von Latenzen und Jitter. Den entstehenden Versatz zwischen Modellannahme und Realität veranschaulicht Abbildung 6.2 (Mitte). Damit einher geht die bereits in Abbildung 6.1 gezeigte unerwünschte Beeinträchtigung der Regelgüte [Ray94, SC95, WNT95] bis hin zum Verlust der Stabilität [HS80]. Zur Lösung dieses Problems bleibt letztlich nur die Verringerung der Abweichungen zwischen Modell und Realität.

Minimierung der zeitlichen Modellabweichungen

Eine gängige Vorgehensweise in der Informatik ist daher der weitgehend modellgetreue Entwurf des Echtzeitsystems und eine damit einhergehende Minimierung der temporalen Abweichungen. Hierbei bedingen die regelungstechnischen Entwurfsannahmen eng bemessene Zeitschranken

und ein strikt periodisches Ausführungsmodell, welches sich beispielsweise durch eine statische Ablaufplanung und ein taktgesteuertes Echtzeitsystem umsetzen lässt [KDK+89, Kop97].

Eine weitere Möglichkeit zur Minimierung des zeitlichen Einflusses der Ausführung liegt in der bereits genannten Überdimensionierung des Rechensystems, beispielsweise durch einen leistungsfähigeren Mikrocontroller oder ein Mehrkernsystem. Diese Maßnahmen sind grundsätzlich zur Lösung des Problems geeignet, konterkarieren jedoch unter anderem mit der Zielsetzung einer rein softwarebasierten und auch nachträglich anwendbaren Fehlertoleranz durch CoRed.

Berücksichtigung der zeitlichen Abweichungen

Sind die zeitlichen Abweichungen zu groß, beispielsweise durch einen entsprechend hohen Anteil der Rechenzeit, ist eine Berücksichtigung der zeitlichen Abweichungen unumgänglich. Aus Sicht der Regelungstechnik manifestieren sich zeitliche Abweichungen als weitere Störungen [SC95] des zu regelnden Prozesses. Eine gängige Vorgehensweise ist daher die Gleichbehandlung zeitlicher wie physikalischer Störungen und ein damit einhergehender angepasster Entwurf der Regelungsanwendung [SC95, ÅW11]. Dieser Ansatz löst das Problem entsprechend durch eine Anpassung der zeitlichen Modellannahmen, zum Beispiel durch eine explizite Berücksichtigung der *Antwortzeit* (engl. *response time*) bis zur Ausgabe des Stellwerts. Im Kontext von Regelungssystemen wird die Zeit zwischen der Auslösung der Aufgabe und der Bereitstellung des Ergebnisses auch als ausführungsbedingte *Totzeit* (engl. *dead time*) bezeichnet.

Ein typisches Beispiel der Modellanpassung ist die Ausgabe der Stellwerte zu einem definierten, späteren Zeitpunkt, zum Beispiel zum Ende der Regelperiode. Dieses Vorgehen ist in vielen Fällen zweckmäßig, es führt jedoch zu gewissen Einbußen bei der erzielbaren Regelgüte und ist daher für dynamische Systeme häufig keine adäquate Option. Im Allgemeinen ist davon auszugehen, dass die schnellstmögliche Ausgabe der Stellwerte im Sinne des Regelungsentwurfs ist. Darüber hinaus hat die verzögerte Ausgabe keinerlei Einfluss auf die zeitlichen Abweichungen an den Eingängen.

6.1.2 Auswirkungen der Replikation

Grundsätzlich entspricht die in Kapitel 3.7.2 vorgeschlagene statische Komposition einer CoRed-Anwendung der Forderung eines möglichst modellgetreuen Entwurfs im Sinne eines taktgesteuerten Echtzeitsystems. Die Steigerung der *Zuverlässigkeit* steht durch die Vervielfachung der Laufzeit jedoch in Wechselwirkung mit der *Rechtzeitigkeit*. Ein Extremfall stellt die Überschreitung der vorgegebenen Regelperiode dar. Hierbei können die gesetzten Termine nicht mehr eingehalten werden und das replizierte Aufgabensystem verliert seine Planbarkeit. In diesem Fall ist die Auslegung des Rechensystems schlicht unzureichend. Unabhängig davon hat die Replikation jedoch in jedem Fall negative Auswirkungen auf die Eigenschaften der Regelung, wie sie bereits in dem Eingangsbeispiel aus Abbildung 6.1 exemplarisch aufgezeigt wurden. Die Art und Schwere der Beeinflussung der Regelgüte ist dabei hochgradig anwendungs- und systemspezifisch. Sie korreliert im Allgemeinen jedoch mit dem Grad der Diskrepanz zwischen regelungstechnischem Modell und Ausführungsrealität [SC95, Tör98].

Relevante zeitliche Eigenschaften

Neben der eigentlichen Regelperiode sind für den gängigen zeitdiskreten Reglerentwurf zwei weitere Eigenschaften der Ausführung von entscheidender Bedeutung: Zum einen ist dies die Antwortzeit bis zur endgültigen Ausgabe des Stellwerts an den Aktor [Tör98, KG02]. Zum anderen umfasst dies jedoch auch eine konsistente Sicht auf die Umwelt. Die durch die Abtastung der Sensorik erfasste *Echtzeitdatenbasis* (engl. *real-time data base*) soll aktuell und nach Möglichkeit

keine zeitlichen Abweichungen zwischen den einzelnen Werten aufweisen. Abbildung 6.2 (unten) zeigt die Beeinflussung beider Aspekte durch die Replikation. Zunächst vergrößert sich sowohl die Zeitspanne d_A bis zur Ausgabe des Stellwerts wie auch die Spanne d_E bis zur vollständigen Bereitstellung der Echtzeitdatenbasis. Die Replikation auf der Eingangsseite führt hierbei zu einem zeitlichen Versatz von Δd_E zwischen den nunmehr redundanten Sensorwerten ($d_{E1} - d_{E3}$). Im Falle eines ressourcenschonenden, optimistischen Duplexbetriebs mit einem Bereitschaftsreplikat, ergibt sich darüber hinaus auch an der Ausgangsseite ein dynamischer zeitlicher Versatz (Δd_A) zwischen Normalbetrieb und Fehlerfall in Form von Jitter.

Kompensation zeitlicher Abweichungen in der Regelung

Die entstehenden konstanten zeitlichen Abweichungen lassen sich zwar, wie bereits erwähnt, im Reglerentwurf relativ einfach berücksichtigen [FPW97, ÅW11], erfordern jedoch die Erweiterung der Anwendung um zusätzliche Regelungselemente für die Kompensation der entsprechenden Totzeiten. Diese Elemente können bereits im regulären Entwurf enthalten sein, sofern die Ausführungszeit auch im unreplizierten Fall relevant ist. Hier genügt prinzipiell die Anpassung der entsprechenden Zeitkonstante, um die Totzeit bis zur Stellwertausgabe auch im replizierten Fall miteinzubeziehen. Im Allgemeinen ist eine geeignete Behandlung der durch die Rechenzeit entstehenden Totzeiten auf einem Rechensystem eher unüblich beziehungsweise in hohem Maße anwendungs- und systemspezifisch [Tör98]. Dies gilt im Besonderen auch für die zeitlichen Abstände innerhalb der Echtzeitdatenbasis.

Insgesamt stellt die Herabsetzung der Regelgüte ein generelles Problem der Replikation dar. Zu dessen Lösung ist eine geeignete Berücksichtigung der zeitlichen Eigenschaften der Ausführung in der Regelungsanwendung zweckmäßig. Dies bedingt in der Regel jedoch die Anpassung und Erweiterung der bestehenden Anwendung.

Fehlende Rückkopplung

Die Annahmen des regelungstechnischen Entwurfsmodells an das temporale Verhalten des Rechensystems sind auf Betriebssystemebene weder offensichtlich noch existieren in den gängigen Entwurfsverfahren geeignete Schnittstellen um dieses zu beeinflussen. Insgesamt erscheint die fehlende Rückkopplung der zeitlichen Auswirkungen der Replikation zwischen Regelungsanwendung und Ausführungsumgebung problematisch.

6.1.3 Verschmelzung zeitlicher Domänen

Ein weiteres Problem ergibt sich aus der technischen Umsetzung der Regelungsanwendung durch die gängigen Entwurfswerkzeuge wie MATLAB/SIMULINK, welche – wie im Falle der Fluglageregelung des I4*Copter*s – typischerweise aus lediglich einer Funktion für den Regelschritt besteht [Tew02]. Hintergrund ist der üblicherweise angestrebte ganzheitliche Entwurf der Regelungsanwendung, welcher alle Teile von der Sensorik bis zur Aktorik einschließt. Diese holistische Betrachtung erlaubt eine globale Optimierung der Signalverarbeitung und Regelung, führt jedoch häufig zu einer Verschmelzung unterschiedlicher zeitlicher Domänen. Die zeitlichen Eigenschaften orientieren sich dabei vorrangig an der Objektdynamik. Sowohl die Signalverarbeitung als auch die Regelung werden anschließend an der hierfür geeignetsten Periode ausgerichtet. Vereinfacht ausgedrückt richten sich sowohl der Entwurf als auch die Ausführung der Regelungsanwendung nach der kleinsten Periode innerhalb des Regelsystems. Konzeptionell eigenständige Bereiche wie die

Sensoren oder die Signalverarbeitung, Filterung und Fusion bilden somit einen integralen Teil des Gesamtmodells und werden durch die Anwendung geeignet koordiniert [KLR94, Tew02, ÅW11]. Entsprechend grobgranular ist jedoch auch die Abbildung auf die Aufgaben des Echtzeitsystems.

Grobgranulare Umsetzung der Regelungsanwendung

Für die Replikation hat dieses Vorgehen jedoch den gravierenden Nachteil, dass das genaue zeitliche Verhalten der Regelungsanwendung unbekannt ist. Die maximale Ausführungszeit (WCET) umfasst folglich alle möglichen Teilaufgaben der Regelung, unabhängig davon, ob diese tatsächlich in einem Zyklus der Regelung zusammenfallen. Eine pessimistische Abschätzung der Laufzeiten und damit auch der Ablaufplanung ist jedoch problematisch. Zum einen führt dies zu einer weiteren Vergrößerung der Antwortzeit beziehungsweise der Zeitdauer d_A bis zur Ausgabe des Stellwerts an den Aktor. Zum anderen steigen durch die monolithische Umsetzung der Regelungsanwendung und die damit verbundene Größe der Prozessinkarnationen die Fehlererkennungslatenzen. Letztlich erschwert die aus Sicht des Betriebssystems nicht zerlegbare Struktur der Regelungsanwendung eine Verfeinerung der Modellannahmen, da kein direkter Zusammenhang zwischen den Modellelementen der Regelung und den Aufgaben des Echtzeitsystems beziehungsweise deren Abbildung auf Prozessinkarnationen existiert. Eine individuelle Zuordnung des zeitlichen Versatzes, beispielsweise zu den Eingängen, ist daher nicht ohne weiteres möglich.

Monolithische Umsetzung der Regelungsanwendung

Die naive Implementierung der Regelungsanwendung verschmilzt die zeitlichen Domänen der einzelnen Teilaufgaben für die Sensorik, Regelung und Aktorik. Dies kann zu einer deutlich pessimistischeren Abschätzung der maximalen Ausführungszeit und einer damit verbundenen, zusätzlichen Vergrößerung der zeitlichen Abstände führen. Zudem existiert für die Rückkopplung der Ausführungszeit kein direkter Zusammenhang zwischen Prozessinkarnation und Modellelementen der Regelung.

6.1.4 Grundannahmen

Der Fokus dieser Arbeit liegt auf der effektiven und weitgehend transparenten Replikation von Regelungsanwendungen. Durch die Wechselwirkung zwischen der Zuverlässigkeit und dem Zeitverhalten erweitert sich die Betrachtung nunmehr auf die Regelungsanwendung selbst. Das Ziel ist letztlich die Bereitstellung eines für die Replikation optimal geeigneten Prozesssystems der Anwendung. Des Weiteren soll die Rückkopplung der veränderten zeitlichen Ausführungseigenschaften an die Regelung eine geeignete Kompensation ohne Änderungen des regelungstechnischen Entwurfs ermöglichen. Im weiteren Verlauf ergeben sich hierfür die folgenden Grundannahmen.

Eigenschaften der Regelungsanwendung

Die Regelungsanwendung ist nach den gängigen Entwurfsprinzipien für die zeitdiskrete Regelung entworfen und entsprechend auf eine festgelegte Regelperiode optimiert. Das Regelungsmodell umfasst dabei alle Teile der Regelstrecke, von der Sensorik und Signalverarbeitung über die eigentliche Regelung bis zur Ausgabe des Stellwerts. Die zeitlichen Domänen der einzelnen Teilaufgaben der Regelung sind jedoch weiterhin sichtbar und erlauben somit grundsätzlich eine Modularisierung des Regelungsmodells unter zeitlichen Gesichtspunkten.

Beachtung der zeitlichen Abweichungen

Eine weitere Grundvoraussetzung ist die prinzipielle Replizierbarkeit der Regelungsanwendung auf einem gegebenen Rechensystem. Die redundante Ausführung soll entsprechend lediglich zu einer handhabbaren Herabsetzung der Regelgüte führen und sich insbesondere innerhalb der gegebenen Regelungsperiode ausführen lassen. Die Berücksichtigung der zeitlichen Abweichungen im Zuge des Anwendungsentwurfs ist zudem zielführend und erlaubt eine ausreichende Kompensierung der negativen Effekte der Replikation.

Hierfür gängige regelungstechnische Entwurfsmethoden umfassen beispielsweise die Einbringung sogenannter *Totzeitglieder* (engl. *lagging element*) oder auch die allgemeine Steigerung der Robustheit gegenüber Störungen der Regelung [FPW97, ÅW11].

6.2 CoRed@Control: Anwendungsschnittstelle

Die vorangegangene erweiterte Problemanalyse zeigt die Wechselwirkungen zwischen den beiden nicht-funktionalen Eigenschaften *Zuverlässigkeit* und *Rechtzeitigkeit* auf. Auf der einen Seite ist CoRed in der Lage die Zuverlässigkeit hinsichtlich der Auswirkungen transienter Fehler deutlich zu verbessern. Andererseits kann die damit verbundene Vervielfachung der Ausführungszeit einen negativen Einfluss auf die Qualität der Regelung haben. Die Ursache hierfür liegt in der Abhängigkeit der Regelungsanwendung von der Zeit beziehungsweise der Abweichung zwischen den regelungstechnischen Annahmen und der tatsächlichen Ausführungsbedingungen. Um die von CoRed angestrebte Transparenz dennoch zu erreichen, ist folglich eine *qualitätsgewahre Replikation* unter Anpassung beziehungsweise Minimierung der Modellabweichungen notwendig. Die spezifische Erweiterung des CoRed-Ansatzes zur CoRed@Control gibt dem Anwendungsentwickler ein hierfür geeignetes Rahmenwerk an die Hand. Das Ziel ist dabei zum einen die Anreicherung der Anwendung um die durch die Replikation entstehenden Zeitabstände. Zum anderen die Modularisierung der Regelungsanwendung und der inhärent verschmolzenen zeitlichen Domänen um eine feingranulare Abbildung auf Prozessinkarnationen und damit eine entsprechend reduzierte Fehlererkennungslatenz zu ermöglichen.

Übersicht und Entwurfskonzept

Bis zu diesem Punkt besteht die einzige Verbindung zwischen der Regelungsanwendung und der CoRed-Dienstschicht beziehungsweise der Ausführungsumgebung im Allgemeinen aus der Aufrufschnittstelle (`set()`, `step()` und `get()`) der Anwendung auf der einen Seite und den Systemaufrufen und Treibern auf der anderen Seite. Der Datenaustausch bezieht sich dabei ausschließlich auf Messwerte und Berechnungsergebnisse. Aus diesem Blickwinkel besteht die Anwendungsschnittstelle bislang lediglich aus einem Datum.

Ein erster logischer Schritt für die angestrebte Rückführung und Kompensation der zeitlichen Abweichungen in der Regelung ist daher die Einbeziehung des *zeitlichen Versatzes* als weiterer Schnittstellenparameter. Dieser entsteht im Zuge der Erfassung der Umwelt durch die Sensoren an den jeweiligen Treibern und erlaubt die Quantifizierung der tatsächlichen Abtastzeitpunkte (d_E). In ähnlicher Weise lässt sich auch der effektive Ausgabezeitpunkt (d_A) ermitteln und bereitstellen.

Zeitliche Abweichung bringt Unsicherheit

Aus regelungstechnischer Sicht zeigen sich die zeitlichen Abweichungen in der Abtastung und Signalverarbeitung jedoch nur mittelbar. Die entscheidende Größe für die Bewertung eines Messsignals ist hier die *Messunsicherheit* (engl. *measurement uncertainty*) [BR02]. Diese beschreibt

letztlich stochastisch den Fehler zwischen dem gemessenen und dem tatsächlichen Wert einer Messgröße. Vereinfacht ausgedrückt korreliert das Rauschen eines Sensors mit der Messunsicherheit der resultierenden Messwerte. Ein zeitlicher Versatz zwischen dem erwarteten und dem realen Abtastzeitpunkt führt entsprechend ebenfalls zu einem Rauschverhalten, welches sich in der Unsicherheit des Messsignals niederschlägt. Die Messunsicherheit spielt in der Signalverarbeitung und Filterung eine entscheidende Rolle und wird beispielsweise bei der Sensorfusion für die Gewichtung der Eingangssignale genutzt. Hierdurch kann sich die Messunsicherheit eines konsolidierten Signals im weiteren Verlauf der Signalverarbeitung entsprechend verändern: Die Verschmelzung zweier unsicherer Werte ergibt einen insgesamt verlässlicheren Messwert.

CORED@CONTROL erweitert entsprechend die Anwendungsschnittstelle um die Messunsicherheit als weiteren Parameter. Diese dient hierbei insbesondere zur Verknüpfung der auf die eigentliche Signalerfassung folgenden Verarbeitungsschritte wie die Sensorfusion oder Beobachter. Eine explizite Modellierung und Berücksichtigung der zeitlichen Eigenschaften ist somit im weiteren Verlauf der Regelung nicht mehr zwingend notwendig, wodurch die im Folgenden näher beschriebene Modularisierung der Regelungsanwendung unterstützt wird.

Anwendungsgewahre, regelungstechnische Schnittstelle

Erst die Erweiterung der Schnittstelle über den reinen Datenaustausch hinaus, ermöglicht die Rückkopplung der Ausführungsbedingungen und damit einen Einfluss auf Modellannahmen der Regelung. Die vorgeschlagene Schnittstelle umfasst hierfür die *Totzeit* und die *Messunsicherheit*. Letztere ist ein entscheidender Wert zur Beschreibung des Messfehlers und dient zur Verknüpfung der zeitinvarianten Berechnungsschritte der Signalverarbeitung.

6.3 Modularisierung und Regelungsarchitektur

Von der Systemebene kommend erlaubt die im vorherigen Abschnitt definierte regelungstechnische Schnittstelle die Verknüpfung der Anwendungselemente in funktionaler und zeitlicher Weise. Die Schnittstelle ist dabei zunächst nur ein Schritt hin zu einer Rückkopplung der zeitlichen Eigenschaften der Ausführung an die Regelungsanwendung. Zusätzlich müssen die entsprechenden Elemente der Signalverarbeitung und Regelung für die Ausführungsumgebung auch zugänglich sein. Die Modularisierung der aus Sicht des Betriebssystems typischerweise zusammenhängenden, monolithischen Regelungsanwendung ist daher der nächste Schritt.

6.3.1 Entwurfskonzept

Die Modularisierung von Regelungsanwendungen stellt im Allgemeinen kein grundlegendes Problem dar. Sie steht jedoch nicht im Mittelpunkt des gängigen Regelungsentwurfs, wie das Fallbeispiel der Fluglageregelung des I4*Copter*s anschaulich zeigt. Die Ursachen liegen in der typischerweise ganzheitlichen Problembetrachtung, welche eine gemeinsame zeitliche Basis für die optimale Abstimmung der regelungstechnischen Aktivitäten anstrebt [KG02, ÅW11].

Entgegen diesem gängigen Vorgehen untergliedert CORED@CONTROL den Regelungsentwurf in logische Schichten. Im Umkehrschluss zu der von der Systemebene bereitgestellten anwendungsgewahren Schnittstelle, zielt die im Folgenden beschriebene Einteilung auf eine *systemgewahre Erweiterung der Regelungsanwendung* ab. Hierfür sind gegebenenfalls über den bestehenden Entwurf hinaus zusätzliche Elemente für die Kompensation des zeitlichen Versatzes notwendig, beispielsweise in Form von Totzeitgliedern. Die hiervon betroffenen Teile befinden sich in der

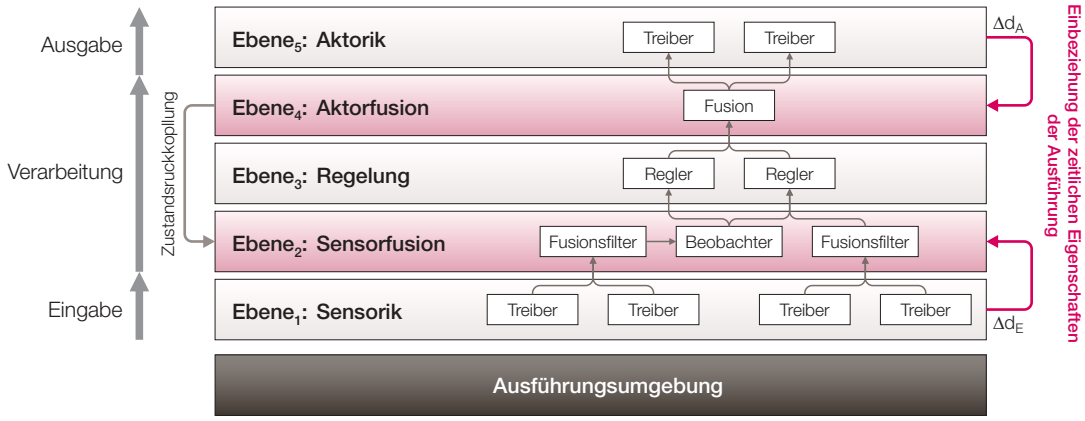


Abbildung 6.3: Schichtenmodell der im I4*Copter* realisierten, modularen Regelungsarchitektur.
Die Architektur dient in erster Linie der Modularisierung der Regelungsanwendung zum Zweck der feingranularen Abbildung auf replizierbare Prozessinkarnationen. Zur Abkopplung der zeitkritischen Eingabe und Ausgabe führt die Architektur *Fusionsebenen* für die Sensorik und Aktorik ein. Deren spezifische Anwendungsschnittstelle erlaubt darüber hinaus die Rückkopplung des ausführungsbezogenen, *zeitlichen Eigenschaften*. Auf der Aktorseite übernimmt die *Zustandsrückkopplung* die Rückführung des Ausgangsversatzes an die Signalverarbeitung.

Signalverarbeitungskette und betreffen hierbei vorrangig die Sensorfusion. Das Ziel der logischen Aufteilung ist es die entsprechenden Eingriffspunkte explizit zu machen, um funktionale Änderungen an der Anwendung im Zuge der Replikation weitgehend überflüssig zu machen. Aufgrund der angestrebten Rückführung der zeitlichen Eigenschaften sowohl bei der Abtastung der Sensorik als auch bei der Ausgabe der Stellwerte an die Aktoren, ist auch eine geeignete Modularisierung der Reglerausgänge in Form einer dedizierten Schicht notwendig.

6.3.2 Schichtenmodell der CoRed@Control-Regelungsarchitektur

Die für die Modularisierung vorgeschlagene Architektur ist in Abbildung 6.3 dargestellt. Sie verfügt neben der bekannten Einteilung in *Eingabe*, *Verarbeitung* und *Ausgabe* über zwei weitere Ebenen. Diese *Fusionsebenen* ermöglichen erst die Abkopplung der zeitlich relevanten Sensorik beziehungsweise Aktorik von der Ausführung der eigentlichen Regelungsanwendung. Insgesamt ergeben sich die folgenden fünf Ebenen:

Sensorik

Diese Ebene entspricht der bekannten Eingabe und sammelt die Daten von den individuellen Sensoren ein. Die Ausführung der eigentlichen Sensortreiber sorgt letztlich für den Aufbau der Echtzeitdatenbasis und bestimmt damit den zeitlichen Versatz der Eingaben Δd_E. Die entsprechenden Informationen werden an den jeweiligen Sensorwert angefügt und somit der nächsten Ebene zugänglich gemacht. Mit der Verfügbarkeit der zeitlichen Informationen ist die Ausrichtung der Abtastperiode an der größten Bandbreite der Sensoren nicht mehr angezeigt, wodurch letztlich die Verschmelzung der zeitlichen Domänen vermieden wird. Stattdessen können die Sensoren mit der jeweils optimalen Abtastrate betrieben werden ohne die Aussagekraft der Echtzeitdatenbasis zu gefährden.

Diese aus Sicht des Systementwurfs vorteilhafte Abtrennung der Sensorik von der weiteren Signalverarbeitung und Sensorfusion erfordert jedoch zusätzliche Maßnahmen im regelungstechnischen Entwurf. Die Sensortreiber müssen für die Einhaltung des Abtasttheorems um geeignete

Filter für die Ratenangleichung (engl. *rate conversion filter*) erweitert und auf die Periode der jeweils nachfolgenden Regelungsblöcke eingestellt werden. An dieser Stelle fügt sich die Messunsicherheit als Schnittstellenparameter ein, da sich diese in Abhängigkeit von dem Grad der Ratendifferenz und der damit verbundenen Überabtastung die Messunsicherheit verändert.

Grundsätzlich ist die Ratenanpassung nicht unkritisch, da die enthaltene Messwerthistorie potenziell zu unerwünschten Wechselwirkungen mit der Regelung führen kann [UFH+12]. Ein problematisches Verhalten ergibt sich jedoch nur bei einem inhomogenen Fehlerverhalten des eingesetzten Filters. Dieses stellt somit kein grundsätzliches Entwurfsproblem dar und kann durch eine geeignete Wahl des Filtertyps gänzlich vermieden werden, zum Beispiel durch den Einsatz von Luenberger-Beobachtern [Lue64].

Sensorfusion

Die Sensorfusion aggregiert und verarbeitet die von der Sensorebene kommenden Messwerte und stellt die konsolidierten physikalischen Messgrößen der Regelung zur Verfügung. Die Ebene nimmt somit die Anwendungskomponenten der Signalverarbeitung auf. Dies umfasst insbesondere die Fusionsfilter für die Konsolidierung verschiedener Sensorwerte. Über die Aufteilung der Signalverarbeitung im I4*Copter*-Fallbeispiel in Kapitel 5.2.2 hinaus, gliedern sich auch die Beobachter für die Rekonstruktion der nicht beobachtbaren Messgrößen auf dieser Ebene ein.

Die von der Sensorik bereitgestellten zeitlichen Informationen werden nun für die Anpassung der Messunsicherheit der jeweiligen Werte genutzt. Hierdurch stimmen die Modellannahmen der Fusion beziehungsweise Zustandsrekonstruktion mit den tatsächlichen Ausführungsbedingungen überein. Ein durch die Replikation entstehender zeitlicher Versatz lässt sich somit ebenso berücksichtigen, wie voneinander abweichende Abtastraten der einzelnen Messquellen. Aus regelungstechnischer Sicht sind die Elemente der Signalverarbeitung um Schnittstellen für die entsprechende Berücksichtigung der zeitlichen Abweichungen und der intern ohnehin verwendeten Messunsicherheit zu erweitern.

Regelung

Am Ende der Sensorfusion steht eine konsistente und den zeitlichen Gegebenheiten angepasste Echtzeitdatenbasis der beobachtbaren wie nicht-beobachtbaren Meßgrößen bereit. Diese werden nun für die Berechnung der eigentlichen Regelungsalgorithmen auf der Regelungsebene herangezogen. Durch die Auftrennung der Regelung lassen sich die einzelnen Teilregler wiederum mit der für sie optimalen Ausführungsperiode betreiben. Im Falle der Fluglageregelung des I4*Copter*s aus Kapitel 5, lassen sich beispielsweise die Regler für die Fluglage und die Flughöhe nunmehr getrennt voneinander betrachten. Durch die deutlich geringe Objektdynamik für die Flughöhe, ergibt sich beispielsweise eine deutlich niedriger Ausführungsperiode von 25 Millisekunden für den Höhenregler. Die Periode der Regelung bestimmt indirekt auch die Ausführungsperiode der daran angeschlossenen Beobachter und Fusionsfilter auf der vorangegangenen Ebene.

Aktorfusion

Analog zur Sensorfusionsebene abstrahiert die Aktorfusion von der eigentlichen Ansteuerung der Aktoren. Diese Signalnachverarbeitung auf dieser Ebene aggregiert auf der einen Seite die Stellwerte der nunmehr separierten Regler und verteilt die kumulierten Werte auf der anderen Seite auf die entsprechenden Aktoren. Eine wichtige Funktion übernimmt die Aktorfusion hinsichtlich der Rückkopplung der zeitlichen Eigenschaften am Ausgang, da diese erst in diesem letzten Verarbeitungsschritt vor der Ausgabe der Stellwerte feststeht. Hinzu kommt die für die Beobachter

essentielle Zustandsrückführung der Stellwerte zurück auf die Ebene der Sensorfusion.

Aus regelungstechnischer Sicht erfordert diese zusätzliche Fusionsebene zum einen die Abtrennung der Aktorfusion von den eigentlichen Reglern. Zum anderen ist auf eine statisches Eingabe-Ausgabe-Verhalten zu achten, um unerwünschte Wechselwirkungen mit der Regeldynamik zu vermeiden. Da sich die Stellwerte der gemeinsam auf einen Aktor wirkenden Regler üblicherweise aufaddieren, ist dieses Verhalten in der Regel gegeben. Die Ausführungsperiode der Elemente der Aktorfusion ergibt sich wiederum aus der minimalen Periode der auf die Fusion wirkenden Regelungskomponenten.

Aktorik

Den letzten Schritt bildet die Aktorikebene. Diese beherbergt die eigentlichen Treiber für das Setzen der Stellwerte an den Aktoren. Als finalen Punkt in der Verarbeitungskette bilden die Treiber der Aktorebene die logische Quelle der zeitlichen Verzögerungen an den Ausgängen Δd_A.

> **Systemgewahrer Regelungsentwurf**
>
> Erst durch die geeignete Rückmeldung der zeitlichen Veränderungen an die Regelungsanwendung ist deren nachgelagerte Replikation möglich. Die Voraussetzung hierfür ist die analoge Berücksichtigung der Ausführungsumgebung im regelungstechnischen Entwurf. Die von CoRed@Control genutzte *Regelungsarchitektur* macht die hierfür notwendigen Eingriffspunkte explizit.

6.4 Modularisierung der Fluglageregelung des I4*Copter*s

Um die grundsätzliche Eignung der durch CoRed@Control vorgeschlagenen Anwendungsschnittstelle und Regelungsarchitektur zu demonstrieren, wurde die Fluglageregelung des I4*Copter*s in geeigneter Weise angepasst.

Erweiterung der CoRed-Dienstschicht

Den ersten logischen Schritt bei der Umsetzung stellt die Erweiterung der CoRed-Dienstschicht dar. Zu diesem Zweck wurden die Treiber für die Sensoren um Bereitstellung des zeitlichen Versatzes (`getDelay()`) erweitert. Der hierfür notwendige Startzeitpunkt entspricht der im Regelungsentwurf angenommenen Phase der Sensorabtastung und ist typischerweise null. Für die Berechnung des zeitlichen Versatzes sieht die Dienstschicht entweder die Übernahme der entsprechenden Phase von der statischen Ablaufplanung im Zuge des Entwurfs- beziehungsweise Erstellungsprozesses vor. Alternativ wird der Zeitabstand zur Laufzeit durch eine Abfrage des Systemzeitgebers berechnet. Zusätzlich wurde eine Ratenanpassung in der allgemeinen Treiberinfrastruktur vorgesehen, welcher die Messunsicherheit (`getUncertainty()`) für den aktuellen Messwert zurückliefert. Für die Einbringung der notwendigen Filter nutzen die Hüllenklassen der Treiber eine Einschubmethode (engl. *hook*), welche durch den Anwendungsentwickler geeignet instrumentiert werden muss.

Modularisierte Fluglageregelung

Die weiteren Änderungen betreffen die Regelungsanwendung und umfassen zunächst die allgemeine Anpassung der Programmcodegenerierung in MATLAB/SIMULINK und die Bereitstellung von entsprechenden Schnittstellen für das Setzen der Verzögerung (`setDelay()`) beziehungsweise das Setzen und Abfragen der Messunsicherheit (`set/getUncertainty()`).

Aus regelungstechnischer Sicht erfordert die weitere Aufteilung des Regelungsmodells eine geringfügig andere Herangehensweise an den Entwurf. Dies betrifft in erster Linie die Aufrechterhaltung der vorgeschlagenen Anwendungsschnittstelle zwischen den einzelnen Elementen der Regelung. An vielen Stellen bedeutet dies jedoch lediglich die explizite Darstellung der im Modell ohnehin vorhandenen Verbindungen zwischen Sensorfusion, Beobachtern und Reglern, so dass diese auch im generierten Quellprogramm noch sichtbar und zugreifbar sind. Ausgehend von der Sensorfusionsebene (vgl. Abbildung 6.3, Ebene_2) teilt sich die bisherige Signalverarbeitung auf. Die bisher in den jeweiligen Fusionsblöcken durchgeführte Ratenanpassung wandert eine Ebene tiefer zu den Sensortreibern. Auf die alternative Signalisierung der Totzeit durch den Sensor konnte im konkreten Fall verzichtet werden. Entsprechend erhält die Sensorfusion im I4*Copter* von der Sensorik stets um die Messunsicherheit angereicherte und vorverarbeitete Messwerte. Die Aufteilung in die drei Sensorgruppen bleibt hierbei bestehen. Zusätzlich sind die Beobachter für die Drehrate, den Winkel und die Höhe nunmehr logisch von der Regelung abgekoppelt und Teil der Sensorfusionsebene. Als Gegenstück zur Sensorfusion übernimmt die Aktorfusionsebene (Ebene_4) die Abtastung der Reglerausgänge und die Fusion der unterschiedlichen Sollwerte zu einem konsolidierten Stellwert für die Übergabe an die Aktortreiber. Die entsprechende Komponente zur Motorsteuerung stellt den einzigen notwendigen Neuentwurf im Rahmen der Modularisierung dar. Als letzter Teil der Signalverarbeitung vor der Ausgabe, dient die Aktorfusion auch der Kompensation des zeitlichen Versatzes am Ausgang. Logisch betrachtet, erhält sie die entsprechende Information von den Aktortreibern. In der technischen Umsetzung erfolgt die Signalisierung durch die Dienstschicht im Zuge der Ausführung. Mit der Abspaltung der Beobachter und der Aktorsteuerung, verbleiben auf der Regelungsebene (Ebene_3) die nunmehr voneinander unabhängigen Regler. Im I4*Copter* umfasst dies den essentiellen Lageregler sowie den Höhenregler.

Insgesamt erforderte die Transformation der regulären, monolithischen Regelungsanwendung des I4*Copter*s lediglich geringfügige Anpassungen an den eigentlichen Regelungsentwurf. Die notwendigen Elemente der Signalverarbeitung, Fusion und die Beobachter waren bereits vorhanden und mussten lediglich logisch voneinander abgekoppelt werden. Durch die Bereitstellung der expliziten Schnittstellen für die Rückführung der zeitlichen Eigenschaften ist eine funktionale Änderung der Regelungsanwendung im Zuge der Replikation im Allgemeinen nicht mehr notwendig. Konstante Totzeiten lassen sich im Zuge des Erstellungsprozesses durch den Regelungsentwurf beziehungsweise die Parameterberechnung des Entwurfswerkzeugs in geeigneter Weise berücksichtigen. Zusätzliche Abweichungen zur Laufzeit lassen sich darüber hinaus durch die Messunsicherheit an die Anwendung weiterreichen. Durch die Aufteilung der Regelungsanwendung in Schichten können die einzelnen Elemente zudem feingranular durch das Entwurfswerkzeug ausgegeben und somit flexibler auf Prozessinkarnationen abgebildet werden. Dies ermöglicht letztlich die Verbesserung der Fehlererkennungslatenz durch die Verkleinerung der Replikate.

Experimentelle Evaluation

Um den Effekt der Modularisierung auf die erzielbare Regelgüte zu untersuchen, wurde der monolithische mit dem modularisierten Fluglageregler vergleichen. Hierfür wurde der I4*Copter* in einen Teststand für den Kippwinkel fixiert und das Verhalten auf eine sprunghafte Änderung der Führungsgröße für den Winkel beobachtet. In beiden Fällen wurden hierbei die zeitlichen

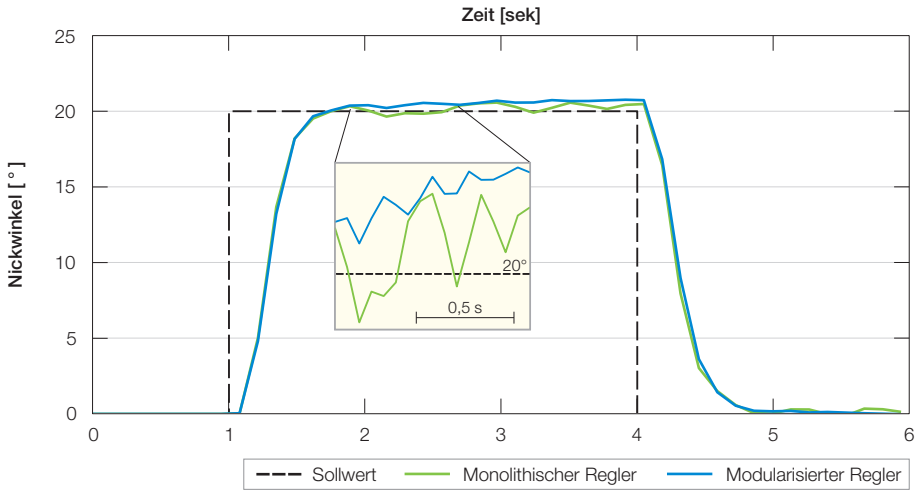


Abbildung 6.4: Vergleich zwischen regulärer und gemäß Regelungsarchitektur modularisierter Fluglageregelung. Vergleichende Messung der Fluglage des I4*Copter* in der kontrollierten Umgebung des Teststands. Der *monolithische Regler* entspricht der regulär entworfenen und nachträglich für die Replikation angepassten Fluglageregelung aus Kapitel 5.1.2. Der *modularisierte Regler* zeigt den geänderten Entwurf gemäß der CoRed@Control-Regelungsarchitektur. Beide Varianten berücksichtigen den zeitlichen Versatz. Im modularisierten Fall wird dieser von den Sensortreibern durch Zeitstempel signalisiert beziehungsweise durch die Ratenanpassung in Messunsicherheit konvertiert. Die Messung zeigt die erwartete Äquivalenz der beiden Varianten hinsichtlich der Regelgüte.

Abweichungen der Ausführung berücksichtigt. Das Ergebnis des Experiments ist in Abbildung 6.4 dargestellt. Es zeigt sich die erwartete Übereinstimmung im Regelverhalten der beiden Varianten.

Verminderung der Fehlererkennungslatenz

Nach der Modularisierung der Regelungsanwendung stehen statt der bisher zwei Aufgaben für die Signalverarbeitung und Regelung potenziell bis zu zehn verschiedene regelungstechnische Aktivitäten für die individuelle Abbildung auf Prozessinkarnationen zur Verfügung. Die augenscheinlichste Änderung ist hierbei die Trennung zwischen Fluglage und Flughöhe. Durch die Abtrennung der Beobachter und der Flughöhe, benötigt die Berechnung des Fluglagereglers beispielsweise statt 187,4 Mikrosekunden nur noch 118,7 Mikrosekunden und somit eine Verminderung der Fehlererkennungslatenz von über 36 Prozent.

6.5 Zusammenfassung

Die weitergehende Betrachtung der softwarebasierten Replikation von Regelungsanwendungen führt letztlich zu der grundsätzlichen Problematik der Wechselwirkung zwischen den nicht-funktionalen Eigenschaften *Zuverlässigkeit* und *Rechtzeitigkeit*. Die durch CoRed angestrebte anwendungstransparente Replikation ist daher in der Praxis mit erheblichen Einschränkungen behaftet.

Die in diesem Kapitel durchgeführte erweiterte Problemanalyse zeigt anschaulich das bei einer *nachträglichen Replikation* auftretende Problem der Diskrepanz zwischen den regelungstechnischen Modellannahmen und der Ausführungsrealität auf. Diese manifestieren sich in Form von zeitlichen Abweichungen bei der Abtastung der Sensoren so wie beim Setzen der Stellwerte und können zu einer signifikanten Herabsetzung der Regelgüte führen. Im Allgemeinen lassen sich die durch die Replikation verändernden zeitlichen Eigenschaften nur durch eine geeignete funktio-

nale Anpassung der Anwendung kompensieren, beispielsweise durch Einbringen entsprechender Totzeit-Elemente. Weiterhin erweist sich die Abbildung der Regelungsanwendung auf Prozessinkarnationen durch den ganzheitlichen Regelungsentwurf als typischerweise grobgranular. Hierdurch ist eine Fehlererkennung erst im Anschluss an die Ausführung der gesamten Regelung möglich, mit entsprechenden Konsequenzen für die erzielbare Fehlererkennungslatenz.

Zur Lösung dieser Probleme bleibt letztlich nur die Minimierung der Modellabweichungen zwischen Regelung und Ausführung. Die Erweiterung von CoRed zu CoRed@Control führt zu diesem Zweck eine anwendungsgewahre Schnittstelle ein, um die geeignete Rückkopplung der zeitlichen Eigenschaften ohne die Notwendigkeit einer funktionalen Anpassung der Regelungsanwendung zu ermöglichen. Diese umfasst neben *Messgröße und Messwert* den *zeitlichen Versatz* der Quelle im Sinne der Eingabelatenz oder alternativ die sich hieraus ergebende *Messunsicherheit*.

Als problematisch für die Rückführung der zeitlichen Abweichungen erweist sich darüber hinaus die durch den ganzheitlichen Regelungsentwurf begünstigte, monolithische Umsetzung der Regelungsanwendung beziehungsweise deren Generierung durch Entwurfswerkzeuge wie MATLAB/SIMULINK. Die für die Rückkopplung relevanten Elemente der Anwendung, wie beispielsweise die Sensorfusion, muss die Regelung daher in geeigneter Weise modularisiert werden. Hierfür nutzt CoRed@Control eine auf diesen Zweck abgestimmte *Regelungsarchitektur*. Diese untergliedert die Anwendung in Schichten für die Sensorik und Aktorik, die Signalverarbeitung und Fusion sowie die eigentliche Regelung. Insbesondere die Elemente der *Sensorfusions-* beziehungsweise *Aktorfusionsebene* übernehmen in diesem Zusammenhang die Rückführung und Kompensation der zeitlichen Abweichungen.

Die Anwendung von CoRed@Control auf die Regelungsanwendung des I4*Copter*s sowie deren Modularisierung erwies sich als weitgehend unproblematisch und erforderte nur minimale Anpassungen an deren Regelungsentwurf. Die notwendigen Änderungen konzentrierten sich im Wesentlichen auf die Einführung der notwendigen Totzeitkompensation sowie die Explizitmachung der Schnittstellen zwischen den Elementen des Modells. Vergleichende Experimente konnten darüber hinaus die Gleichwertigkeit der modularisierten Variante der Fluglageregelung hinsichtlich der erzielbaren Regelgüte demonstrieren.

7 Zusammenfassung und Ausblick

Die Forschungsfrage dieser Dissertation befasst sich mit der systematischen Beeinflussung der nicht-funktionalen Eigenschaft *Zuverlässigkeit* durch softwarebasierte Redundanz auf der Ebene des Betriebssystems. Im Fokus steht hierbei die zuverlässige Erkennung von transienten Fehlern der Hardware. Den Ausgangspunkt bildete das existierende Wissen zur Erkennung von transienten Fehlern durch strukturelle Redundanz und Informationsredundanz. Die Fragestellung stützt sich auf die Grundannahme, dass sich transiente Fehler stets in den Strukturelementen der Software manifestieren und Gleichtaktfehler ausgeschlossen sind. Hinterfragt wird die tatsächliche Zuverlässigkeit der arithmetischen Codierung als einer Redundanztechnik mit konzeptionell vollständiger Fehlererkennung. Im Speziellen wird weiterhin der Zusammenhang zwischen der nachträglichen Replikation von Regelungsanwendungen und deren Regelgüte betrachte.

7.1 Zusammenfassung

Der in dieser Arbeit entwickelte *kombinierte Redundanzansatz* CoRed (engl. *Combined Redundancy*) eliminiert die kritischen Fehlerstellen in softwarebasierten Fehlertoleranz. Der Ansatz basiert auf der namensgebenden Kombination aus der replizierten Ausführung von Prozessinkarnationen auf der Ebene des Betriebssystems sowie der arithmetischen Codierung der verbleibenden Replikationsinfrastruktur und Datenflüsse zwischen den Replikaten. Das zentrale Element bildet in diesem Zusammenhang der zuverlässige CoRed-*Mehrheitsentscheider*. Insgesamt schließt CoRed die bislang unvermeidbaren Zuverlässigkeitsengpässe und agiert somit als *Wächter der Wächter* über die replizierte Ausführung der Anwendung. Durch die gänzliche Vermeidung von kritischen Fehlerstellen ermöglicht CoRed die gezielte Beeinflussung der Zuverlässigkeit der zu schützenden Anwendung. Die in diesen Fällen obligatorische Zuverlässigkeitsbetrachtung des Systems vereinfacht sich hierdurch entscheidend auf die gängigen Verfahren für die Replikation.

Leistungsfähige Fehlererkennung

Die Basis für die Fehlererkennung in den nichtreplizierbaren Teilen der redundanten Ausführung stellt die arithmetische Codierung in Form von ANBD-Codes dar. Deren konzeptionell vollständige

Fehlererkennung kann auf dem Weg von dem theoretischen Entwurf zu der praktischen Realisierung verloren gehen. Die erreichbare Effektivität hängt zum einen von der geeigneten Wahl der Codierungsparameter ab – vergleichbar der Wahl eines guten Passworts. Zum anderen gefährden auch Schwachstellen in der technischen Umsetzung der Codierung eine zuverlässige Fehlererkennung. Die erstmalige, vollständige Evaluation aller 16-Bit-Codierungsschlüssel zeigt eindrucksvoll, dass gängige Empfehlung wie *je größer, desto besser* nicht zwangsläufig zutreffen. Die Auswahl der leistungsfähigsten, für 32-Bit-AN-Codes verfügbaren, Schlüssel erlaubt CoRed die sichere Erkennung von bis zu 5-fach-Bitfehlern. Stille Annahmen hinsichtlich der Abgeschlossenheit des Coderaumes oder bestimmter Konventionen der Binärschnittstelle der Hardwareplattform durch den Compiler oder das Betriebssystem repräsentieren typische Beispiele für Schwachstellen in der Implementierung, welche zu einem signifikanten Verlust der Erkennungsleistung führen können. CoRed eliminiert all diese Schwachstellen der technischen Umsetzung und ist dadurch in der Lage, die arithmetische Codierung bis an ihre Leistungsgrenzen zu führen. Im Vergleich mit gängigen Ansätzen zeigt sich dies in einer deutlichen Verbesserung der Erkennungsleistung um teilweise den Faktor 1000. Insgesamt ermöglicht CoRed nicht nur konzeptionell, sondern auch praktisch die *vollständige* und *zuverlässige* Fehlererkennung.

Umfassender Schutz

Die natürliche Zieldomäne für CoRed stellen aufgrund ihrer Zahl und des ausgewiesenen Schutzbedürfnisses gegenüber transienten Fehlern sicherheitskritische Regelungsanwendungen dar. Die praktische Umsetzbarkeit des CoRed-Ansatzes wurde daher am Beispiel des I4*Copter*s veranschaulicht. Dieses unbemannte Luftfahrzeug erfordert durch seine inhärent instabilen Flugeigenschaften eine kontinuierliche Fluglageregelung und stellt somit ein forderndes und praxisnahes Beispiel dar. Das System basiert auf einem in der Fahrzeugelektronik verbreiteten Infineon TriCore TC1796 Mikrocontroller und umfasst eine umfangreiche Inertial-Sensorik sowie Aktorik. Auch auf dieser Hardwareplattform konnte die vollständige und zuverlässige Fehlererkennung durch die CoRed-Dienstprimitive mittels Fehlerinjektion nachgewiesen werden. Insgesamt demonstriert die erfolgreiche und mit lediglich geringfügigen Anpassungen der Anwendung verbundene Absicherung der Fluglageregelung die allgemeine Umsetzbarkeit von CoRed.

Der mit CoRed verfolgte Fehlertoleranzansatz zielt auf eine ganzheitliche Absicherung der zu schützenden Anwendung ab. Hierfür erstreckt sich der entstehende Redundanzbereich von deren Eingängen bis zu den Ausgängen. Darüber hinaus lässt sich der Schutz auch über die Grenzen des Rechensystems hinaus erweitern. Hierfür wurde der I4*Copter* um ein Versuchsmuster eines codierenden Sensors sowie ein intelligentes Aktorsystem erweitert. Durch die codierte Übertragung der Messwerte können weite Teile der umständlichen Eingangsreplikation ohne den Verlust der Fehlererkennung entfallen. Auf der Ausgangsseite erlaubt die Weitergabe des codierten Gewinners des finalen Mehrheitsentscheids die Ausdehnung des Redundanzbereichs bis auf den Aktor. Durch die nachgelagerte Decodierung kann dieser neben der Datenintegrität auch die korrekte Ausführung des entfernten Mehrheitsentscheiders überprüfen. Der Charme dieser Erweiterbarkeit liegt in dem nahtlosen Übergang zwischen der Datenübertragung, Berechnung und der Konsolidierung der Ergebnisse der redundanten Ausführung durch den Mehrheitsentscheider.

Im Wechselspiel mit der Anwendung

Die weitergehende Betrachtung der softwarebasierten Replikation von Regelungsanwendungen zeigt anschaulich die hierbei auftretenden Wechselwirkungen zwischen den nicht-funktionalen Eigenschaften *Zuverlässigkeit* und *Rechtzeitigkeit*. Die bei der *nachträglichen* Vervielfältigung auftretenden Abweichungen zwischen regelungstechnischen Entwurfsmaßnahmen und der Ausfüh-

rungsrealität können hierbei negative Folgen für die erzielbare Regelgüte haben. Die Erweiterung von CoRed zu CoRed@Control führt daher eine anwendungsgewahre Schnittstelle für die Rückkopplung des zeitlichen Versatzes ein. Darüber hinaus erlaubt eine darauf abgestimmte Regelungsarchitektur die Modularisierung der Regelungsanwendung um die notwendigen Eingriffspunkte von außen zugänglich zu machen. Die Modularisierung dient zudem der feingranularen Abbildung der regelungstechnischen Aufgaben auf replizierbare Prozessinkarnationen und damit zur Verminderung der Fehlererkennungslatenz.

Summa summarum

Zusammenfassend bietet CoRed das Rüstzeug für die gezielte Beeinflussung der Zuverlässigkeit auf der Ebene des Betriebssystems. Darüber hinaus unterstützt CoRed@Control die Absicherung sicherheitskritischer Regelungsanwendungen durch eine anwendungsgewahre Rückkopplung der sich durch die Replikation verändernden zeitlichen Eigenschaften.

7.2 Beiträge dieser Arbeit

Im Mittelpunk dieser Dissertation stand die Bereitstellung einer zuverlässigen und selektiv anwendbaren Fehlertoleranz. Die Entwicklung des CoRed-Ansatzes erstreckt sich jedoch auch auf hieran angrenzende Bereiche, wie beispielsweise die Domäne der Regelungsanwendungen. Die folgende Aufstellung fasst die wichtigsten Beiträge dieser Arbeit zum Stand der Kunst zusammen:

CoRed bietet das Rüstzeug für die systematische Beeinflussung der nicht-funktionalen Eigenschaft Zuverlässigkeit auf der Ebene des Betriebssystems. Der koordinierte Einsatz der arithmetischen Codierung eliminiert erstmalig alle kritischen Fehlerstellen und Zuverlässigkeitsengpässe in der softwarebasierten Replikation. Der Ansatz erlaubt damit den Aufbau eines geschlossenen Redundanzbereichs, welcher sich auch über die Systemgrenzen hinaus erweitern lässt. Durch die Kombination von Codierung und Replikation bietet CoRed somit eine *ganzheitliche Fehlertoleranz für eingebettete Softwaresysteme*.

Eine arithmetische Codierung, welche ihre volle Leistungsfähigkeit in der Praxis ausschöpft. Hierfür verantwortlich ist zum einen die Beseitigung der Schwachstellen in der technischen Umsetzung der Codierung. Zum anderen ermöglicht erst der zweistufige Auswahlprozess der Codierungsparameter die Wahl der tatsächlich leistungsfähigsten und bei einer Überbeanspruchung gutmütigen Codierungsschlüssel und Signaturen.

CoRed@Control erlaubt die anwendungsgewahre Replikation von Regelungsanwendungen durch die Rückführung des hierdurch entstehenden zeitlichen Versatzes. Die darauf aufbauende Modularisierung in die Schichten der CoRed@Control-Regelungsarchitektur sorgt im Umkehrschluss für eine systemgewahre Regelungsanwendung. Durch die feingranulare Abbildung der regelungstechnischen Aufgaben auf Prozessinkarnationen wird zudem die Fehlererkennungslatenz reduziert.

Der I4*Copter* stellt schließlich ein forderndes und realistisches Fallbeispiel für eine sicherheitskritische Regelungsanwendung dar. Sein Steuergerät wurde spezifisch für die Evaluation entsprechender Anwendungsszenarien entwickelt. Darüber hinaus stehen mit dem intelligenten Aktorsystem und dem barometrischen Sensorknoten zwei codierte Peripheriesysteme zur Verfügung. Die Relevanz der Anwendung zeigt sich in dem, über die Verwendung im Rahmen dieser Arbeit hinausgehenden, Einsatz des I4*Copter*s als Fallstudie in einer ganzen Reihe weiterer Forschungsprojekte an der Friedrich-Alexander-Universität Erlangen-Nürnberg.

7.3 Schlussfolgerungen

Im Allgemeinen lassen sich die folgenden, wesentlichen Schlussfolgerungen aus den Ergebnissen und Erkenntnissen dieser Arbeit ziehen:

Zuverlässige softwarebasierte Fehlertoleranz ist möglich. Die Voraussetzung hierfür ist jedoch das Vorhandensein von Redundanz in *jedem* Schritt der Ausführung. Unter dieser Bedingung lassen sich alle transienten Fehler erkennen, die sich in den Strukturelementen der Software manifestieren. Essentiell bei der Umsetzung entsprechender Maßnahmen wie CoRed ist die Betrachtung aller Ebenen vom Entwurf bis zu der plattformspezifischen Befehlssatzebene. Abschließend ist der Ausschluss von Gleichtaktfehlern eine augenscheinlich notwendige Bedingung. Das bedeutet, dass die Hardware keinesfalls großflächig oder gänzlich versagt.

Der gemeinsame Entwurf von Regelungsanwendung und Echtzeitsystem ermöglicht erst die anwendungstransparente Umsetzung softwarebasierter Replikationsansätze wie CoRed. Die Ausführungsbedingungen haben einen signifikanten Einfluss auf die Regelgüte. Durch die anwendungsgewahre Replikation und die geeignete Rückmeldung der zeitlichen Veränderungen ist eine nachgelagerte Replikation der Anwendung möglich. Die Voraussetzung hierfür ist die analoge Berücksichtigung der Ausführungsumgebung im regelungstechnischen Entwurf. Eine geeignete Schnittstelle kann hierbei den Aufwand auf beiden Seiten reduzieren.

Für die Entwicklung robuster Software gibt es wenig Unterstützung durch die gängigen Techniken und Werkzeuge der Softwaretechnik, da diese ein Auftreten transienter Fehler weder annehmen noch berücksichtigen. Stille Annahmen des Betriebssystems beziehungsweise des Compilers hinsichtlich der Hardwareplattform sind eine weitere subtile Fehlerquelle. Die Konventionen der Binärschnittstelle, wie zum Beispiel das Sichern bestimmter Register bei einem Kontextwechsel, basieren stets auf der Annahme eines fehlerfrei arbeitenden Systems und verlieren ihre Gültigkeit im Fehlerfall. Insgesamt zeigten sich die Schwachstellen der Umsetzung ausnahmslos in wenig offensichtlichen und seltenen Randfällen. Die Entwicklung gegenüber transienter Fehler robuster Software profitiert daher nachhaltig von der Unterstützung durch entsprechende Fehlersimulations- oder Fehlerinjektionswerkzeuge.

Die Betriebssystemebene ist prädestiniert für die Umsetzung der softwarebasierten Replikation. Diese erlaubt durch die bereitgestellten Strukturelemente eine einfache Isolation der Replikate. Die sich an der Anwendungsstruktur orientierende Granularität der Prozessinkarnationen erscheint zudem äußerst zweckmäßig hinsichtlich der für die Fehlerdiagnose benötigten Bandbreite. Einen weiteren wichtigen Aspekt stellen die bereits erwähnten stillen Annahmen dar. Diese werden vorrangig durch das Betriebssystem ausgenutzt und können somit direkt an der Quelle angepasst werden.

7.4 Ausblick

Als Abschluss dieser Arbeit dienen die folgenden, aus den Erkenntnissen dieser Dissertation gewonnenen, Ideen und weiterführende Forschungsfragen.

7.4.1 Absicherung der Ausführungsumgebung

Bislang zielt CoRed alleine auf die Verbesserung der Zuverlässigkeit von sicherheitskritischen Anwendungen ab. Im Bezug auf die Ausführungsumgebung setzt CoRed hingegen auf eine absolute Minimierung der Angriffsfläche. Grundsätzlich lassen sich jedoch dieselben Konzepte auch

auf die Komponenten des Betriebssystems selbst anwenden. Ein erster Schritt in diese Richtung stellt bereits die Emulation der Systemschnittstellen dar. Diese übernehmen in ihrer Funktion als Stellvertreter die Codierung und Decodierung der entsprechenden Werte des Systemaufrufs für die Anwendung. Stellt das Betriebssystem seinerseits codierte Systemaufrufe bereit, kann der Stellvertreter entfallen und der Redundanzbereich entsprecht ausgedehnt werden.

Einen ersten Schritt in diese Richtung geht CoRedOS [HDL13], welches die Konzepte von CoRed auf die kritischen Funktionen des Betriebssystems ausweitet und somit eine native, codierte Systemschnittstelle anbieten kann. Bislang erfolgt ein großer Teil der notwendigen arithmetischen Codierung nach wie vor statisch im Zuge des Erstellungsprozesses. Die künftigen Herausforderungen liegen daher vor allem in der Unterstützung dynamischer Anwendungen und Systeme, welche beispielsweise auf eine Koordinierung durch das Betriebssystem zur Laufzeit angewiesen sind.

7.4.2 Werkzeugunterstützung und Automatisierung

In seiner eigentlichen Ausprägung als Dienstschicht zwischen Betriebssystem und Anwendung erfordert CoRed die manuelle Konfiguration der Replikation sowie die Nutzung der entsprechenden CoRed-Dienstprimitive und Treiber durch die Anwendung. Der Vorteil dieses Vorgehens ist die explizite und für den Entwickler sichtbare Umsetzung der Fehlertoleranz. Die Instanziierung der Replikate und die notwendige Parametrierung der Codierung erfolgen im Zuge der Erzeugung des Binärprogramms. Hierfür erweitert und instrumentiert CoRed den regulären Konfigurations- und Erstellungsprozess der ohnehin genutzten Werkzeugkette, im konkreten Fall der GNU-Compilersammlung. Diese absichtliche Beschränkung begründet sich in den geltenden Normen [IEC98, ISO11b] für sicherheitskritische Anwendungen, welche häufig den Einsatz qualifizierter und betriebsbewährter Werkzeuge erfordern.

Unabhängig davon ist eine weitere Automatisierung der Replikation in vielen Anwendungsbereichen wünschenswert. Daher wurde der CoRed-Ansatz seit seiner Konzeption bereits in weiteren Forschungsprojekten an der Friedrich-Alexander-Universität Erlangen-Nürnberg aufgegriffen und in andere Entwicklungsumgebungen überführt [Sch11a, TSK+11].

Anwendungstransformation durch den Real-Time Systems Compiler

In diesem Rahmen wurde auch der Einsatz von CoRed im *Real-Time Systems Compiler* (RTSC) [SSP11], einem Werkzeug zur Manipulation der *Echtzeitsystemarchitektur*, evaluiert. Dieser extrahiert hierfür die Strukturelemente eines Eingangssystems und erzeugt daraus eine architekturneutrale Darstellung, welche geeignet angepasst und mit veränderter Architektur wiederum als Zielsystem ausgegeben werden kann. Das Hauptanwendungsgebiet ist hierbei die Überführung eines ereignisgesteuerten Systems in ein äquivalentes taktgesteuertes Echtzeitsystem. Für die Replikation bietet sich diese Transformation insbesondere wegen des geforderten Replikdeterminismus an, welcher sich in einem taktgesteuerten System vergleichsweise einfach umsetzen lässt.

Der RTSC übernimmt als Eingabe ein Quellsystem, zusammen mit einer Beschreibung von dessen spezifischen Echtzeiteigenschaften in Form eines Systemmodells. Die Replikation einzelner Anwendungsteile erfolgt durch eine geeignete Annotation der Systembeschreibung automatisiert im Zuge des Transformationsprozesses.

Erweiterte Transformation von Regelungsanwendungen

Eine herausragende Eigenschaft des RTSC ist die Analyse der Abhängigkeiten und zeitlichen Eigenschaften des Quellsystems ebenso wie die des erzeugten Zielsystems. Die für CoRed@Control

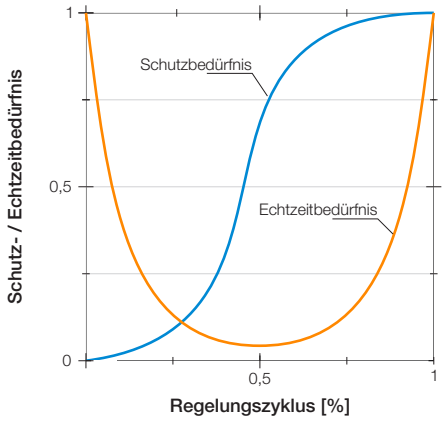


Abbildung 7.1: Inhärente Fehlertoleranz versus zeitliche Anforderungen der I4*Copter* Fluglageregelung.
Dieses schematische Beispiel veranschaulicht die angenommene inhärente Fehlertoleranz der Fluglageregelung im Vergleich zu ihren zeitlichen Anforderungen. Das Schutzbedürfnis steigt im Zuge der Verrechnung der redundanten Messwerte kontinuierlich an. Demgegenüber sind die zeitlichen Anforderungen allem im Zuge des Abtastens beziehungsweise Setzens sehr hoch, während sie in der Berechnungsphase auf einem niedrigen Niveau verbleiben.

erforderlichen zeitlichen Eigenschaften der Regelungsanwendung lassen sich folglich ebenfalls im Rahmen dieses Prozesses gewinnen und den jeweiligen Anwendungskomponenten zuführen. Dies stellt einen vielversprechenden Lösungsweg für die automatisierte Replikation von Regelungsanwendung dar. Vorstellbar ist an dieser Stelle sogar der bidirektionale Informationsfluss, in dem der RTSC auch die Ablaufplanung der Replikate gemäß der Qualitätsanforderungen der Anwendung übernimmt und optimiert.

7.4.3 Hydra – Hybrid-kombinierte Redundanz Architektur

Eine der Grundannahmen dieser Arbeit ist die Abwesenheit von Gleichtaktfehlern der ausführenden Hardwareplattform, da ein Ausfall der Hardware keinesfalls durch die softwarebasierte Fehlertoleranz maskiert werden könnte. In Anwendungsfällen in denen diese Annahme nicht getroffen werden kann oder darf, bietet sich die erneute Kombination von Redundanztechniken an: Die hybride Auslegung von hardwarebasierter und softwarebasierter Replikation. Die herausragende Eigenschaft von CoRed ist die vollständige Vermeidung von unerkannten Datenfehlern. Somit verhält sich ein durch CoRed geschütztes System inaktiv im Fehlerfall. In anderen Worten vereinfacht CoRed das Fehlermodell des Rechensystems durch die Eliminierung transienter Fehler aus der Fehlerhypothese.

Auch ohne ein entsprechendes Verhalten der Anwendung, ist hierdurch eine der Grundvoraussetzungen für den Aufbau eines Duplexsystems gegeben. Die Kombination von software- und hardwarebasierter Redundanz kann somit eine kosteneffiziente Alternative für die sonst notwendige dreifache Hardwareredundanz darstellen. Hierfür notwendig ist jedoch die enge Abstimmung zwischen den codierten Anwendungen, beispielsweise durch einen synchronen Datenbus. Eine sich durch die selektive Anwendbarkeit von CoRed eröffnende Möglichkeit liegt in der heterogenen Auslegung der beiden Rechensysteme. Neben den notwendigen Mechanismen für die *Ausfallsicherung* (engl. *fail over*) im Fehlerfall ist hierbei vor allem die tatsächliche Effektivität eine offene Forschungsfrage.

7.4.4 Ausnutzung der inhärenten Fehlertoleranz von Regelungssystemen

Das Beispiel der Fluglageregelung des I4*Copter*s veranschaulicht die typischerweise inhärente Redundanz entsprechender Anwendungen. Ein transienter Fehler in einem der redundanten Sensortreiber wird sich in aller Regel nicht schwerwiegend auf den Ausgang der Regelung auswirken. Im Zuge der Signalverarbeitung werden die redundanten Messwerte jedoch konsolidiert. Entsprechend steigt das Schutzbedürfnis der Anwendung, wie in Abbildung 7.1 dargestellt, im Verlauf des Berechnungsfortschritts bis zu Stellwertausgabe immer weiter an. Im Vergleich dazu ist die Anforderung an die Rechtzeitigkeit im Zuge der Abtastung der Sensoren sowie bei der Stellwertausgabe besonders hoch.

In beiden Fällen erscheint die Messunsicherheit als geeigneter Gradmesser für die Bestimmung des Einflusses einer Komponente auf das Endergebnis. Die inhärente Redundanz einer Regelungsanwendung lässt sich somit für die ressourceneffiziente Fehlertoleranz ausnutzen. Dies gilt im selben Maß auch für die Ablaufplanung, welche nunmehr ebenfalls abhängig von den Auswirkungen auf die Bedürfnisse der Anwendung durchgeführt werden kann. Eine Stoßrichtung für die Umsetzung eines solchen adaptiven Systems ist die Bereitstellung entsprechender Basismechanismen für die Ablaufkoordinierung auf der Basis der Messunsicherheit durch das Betriebssystem. Die ersten Versuche sind hier vielversprechend.

Literaturverzeichnis

[AAA+90] **Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E. und Powell, D.** (1990): Fault Injection for Dependability Validation: A Methodology and Some Applications. IEEE Transactions on Software Engineering, 16(2): Seiten 166–182. ISSN 0098-5589. doi:10.1109/32.44380. (Zitiert auf Seite 89.)

[ABK+93] **Ackermann, J., Bartlett, A., Kaesbauer, D., Sienel, W. und Steinhauser, R.** (1993): Robust Control: Systems with Uncertain Physical Parameters. 1. Auflage. Springer, New York, NY, USA. ISBN 978-0-3871-9843-9. (Zitiert auf den Seiten 31 und 123.)

[AGM+71] **Avižienis, A., Gilley, G., Mathur, F.P., Rennels, D., Rohr, J. und Rubin, D.** (1971): The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design. IEEE Transactions on Computers, 20(11): Seiten 1312–1321. ISSN 0018-9340. doi:10.1109/T-C.1971.223133. (Zitiert auf den Seiten 37 und 38.)

[ALRL04] **Avižienis, A., Laprie, J.C., Randell, B. und Landwehr, C.** (2004): Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 1(1): Seiten 11–33. ISSN 1545-5971. doi:10.1109/TDSC.2004.2. (Zitiert auf den Seiten 21 und 29.)

[Arn73] **Arnold, T.F.** (1973): The Concept of Coverage and Its Effect on the Reliability Model of a Repairable System. IEEE Transactions on Computers, 22(3): Seiten 251–254. ISSN 0018-9340. doi:10.1109/T-C.1973.223703. (Zitiert auf den Seiten 10 und 48.)

[ASB+08] **Afonso, F., Silva, C., Brito, N., Montenegro, S. und Tavares, A.** (2008): Aspect-oriented Fault Tolerance for Real-time Embedded Systems. Proceedings of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS '08, Seiten 2:1–2:8. ACM Press, New York, NY, USA. ISBN 978-1-60558-142-2. doi:10.1145/1404891.1404893. (Zitiert auf Seite 42.)

[Atm09] **Atmel Corporation** (2009): AVR XMEGA A – Handbuch. Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131, USA. (Zitiert auf den Seiten 120 und 16.)

[AUT11] **AUTOSAR** (2011): Specification of Operating System (Version 5.0.0). AUTOSAR development cooperation, Bernhard-Wicki-Strasse 3, D-80636 München. (Zitiert auf den Seiten 110 und 6.)

[AVFK02] **Aidemark, J., Vinter, J., Folkesson, P. und Karlsson, J.** (2002): Experimental evaluation of time-redundant execution for a brake-by-wire application. Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN '02), Seiten 210–215. IEEE Computer Society Press, Washington, DC, USA. doi:10.1109/DSN.2002.1028902. (Zitiert auf den Seiten 42, 47 und 48.)

[Avi71] **Avižienis, A.** (1971): Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design. IEEE Transactions on Computers, C-20(11): Seiten 1322–1331. ISSN 0018-9340. doi:10.1109/T-C.1971.223134. (Zitiert auf Seite 39.)

[ÅW11] **Åström, K.J. und Wittenmark, B.** (2011): Computer-Controlled Systems: Theory and Design. 3. Auflage. 576 Seiten, Dover Publications. ISBN 978-0-4864-8613-0. (Zitiert auf den Seiten 10, 55, 71, 123, 125, 126, 127, 128, 129 und 130.)

[Bau05] **Baumann, R.C.** (2005): Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability, 5(3): Seiten 305–316. (Zitiert auf den Seiten 3 und 4.)

[BCS69] **Bouricius, W.G., Carter, W.C. und Schneider, P.R.** (1969): Reliability modeling techniques for self-repairing computer systems. Proceedings of the 24th ACM National Conference (ACM '69), Seiten 295–309. ACM Press, New York, NY, USA. doi:10.1145/800195.805940. (Zitiert auf den Seiten 10 und 48.)

[BDCDNP03] **Benso, A., Di Carlo, S., Di Natale, G. und Prinetto, P.** (2003): A watchdog processor to detect data and control flow errors. Proceedings of the 9th IEEE On-Line Testing Symposium (IOLTS '03), Seiten 144–148. IEEE Computer Society Press, Washington, DC, USA. doi:10.1109/OLT.2003.1214381. (Zitiert auf Seite 32.)

[BDF+03] **Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. und Warfield, A.** (2003): Xen and the Art of Virtualization. Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03), Band 37, 5 von *ACM SIGOPS Operating Systems Review*, Seiten 164–177. ACM Press, New York, NY, USA. doi:10.1145/945445.945462. (Zitiert auf Seite 8.)

[BDGR97] **Bugnion, E., Devine, S., Govil, K. und Rosenblum, M.** (1997): Disco: Running Commodity Operating Systems on Scalable Multiprocessors. ACM Transactions on Computer Systems, 15(4): Seiten 412–447. ISSN 0734-2071. doi:10.1145/265924.265930. (Zitiert auf Seite 8.)

[BG97] **Bodson, M. und Groszkiewicz, J.** (1997): Multivariable adaptive algorithms for reconfigurable flight control. IEEE Transactions on Control Systems Technology, 5(2): Seiten 217–229. ISSN 1063-6536. doi:10.1109/87.556026. (Zitiert auf den Seiten 31 und 123.)

[BGM12] **Braun, J., Geyer, D. und Mottok, J.** (2012): Alternative Measure for Safety-Related Software. ATZelektronik worldwide, 7(4): Seiten 40–43. doi:10.1365/s38314-012-0106-1. (Zitiert auf Seite 97.)

[BKL89] **Brilliant, S., Knight, J. und Leveson, N.** (1989): The consistent comparison problem in N-version software. IEEE Transactions on Software Engineering, 15(11): Seiten 1481–1485. ISSN 0098-5589. doi:10.1109/32.41339. (Zitiert auf Seite 60.)

[BKL+10] **Blanke, M., Kinnaert, M., Lunze, J., Staroswiecki, M. und Schröder, J.** (2010): Diagnosis and Fault-Tolerant Control. 2. Auflage. Springer, Berlin. ISBN 978-3-642-07136-2. (Zitiert auf den Seiten 31 und 123.)

[Bla11] **Blank, P.** (2011): Ein Miniaturquadrokopter als Plattform für mobile, adaptive Sensornetzwerke. Bachelorarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[Bor05] **Borkar, S.** (2005): Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro, 25(6): Seiten 10–16. ISSN 0272-1732. doi:10.1109/MM.2005.110. (Zitiert auf den Seiten 3 und 6.)

[BP03] **Benso, A. und Prinetto, P.** (Hg.) (2003): Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation. Frontiers in Electronic Testing, 1. Auflage. 241 Seiten, Springer, Boston, MA, USA. ISBN 978-1-4020-7589-6. (Zitiert auf Seite 89.)

[BR02] **Bevington, P.R. und Robinson, K.D.** (2002): Data reduction and error analysis for the physical sciences. 3. Auflage. 320 Seiten, McGraw-Hill, Toronto, Canada. ISBN 978-0-0711-9926-1. (Zitiert auf Seite 129.)

[Bre98] **Bressoud, T.** (1998): TFT: A Software System for Application-Transparent Fault Tolerance. Proceedings of the 28th International Symposium on Fault-Tolerant Computing, Seiten 128–137. IEEE Computer Society Press, Washington, DC, USA. ISSN 0731-3071. doi:10.1109/FTCS.1998.689462. (Zitiert auf den Seiten 41 und 42.)

[Bro06] **Broy, M.** (2006): Challenges in Automotive Software Engineering. Proceedings of the 28th International Conference on Software Engineering (ICSE '06), Seiten 33–42. ACM Press, New York, NY, USA. ISBN 1-59593-375-1. doi:10.1145/1134285.1134292. (Zitiert auf Seite 9.)

[Brü05] **Brütting, W.** (2005): Physics of organic semiconductors. 1. Auflage. 658 Seiten, Wiley-VCH, Weinheim. ISBN 978-3-527-40550-3. (Zitiert auf den Seiten 2 und 3.)

[BS96] **Bressoud, T.C. und Schneider, F.B.** (1996): Hypervisor-based fault tolerance. ACM Transactions on Computer Systems, 14(1): Seiten 80–107. ISSN 0734-2071. doi:10.1145/225535.225538. (Zitiert auf Seite 41.)

[BTM95] **Black, J., Taylor, D. und Morgon, D.** (1995): A Compendium of Robust Data Structures. Proceedings of the 25th International Symposium on Fault-Tolerant Computing, Seiten 127–129. IEEE Computer Society Press, Washington, DC, USA. doi:10.1109/FTCSH.1995.532623. (Zitiert auf Seite 33.)

[CMC+13] **Cho, H., Mirkhani, S., Cher, C.Y., Abraham, J. und Mitra, S.** (2013): Quantitative evaluation of soft error injection techniques for robust system design. Proceedings of the 50th ACM / EDAC / IEEE Design Automation Conference (DAC '13), Seiten 1–10. ISSN 0738-100X. (Zitiert auf Seite 99.)

[CMDD62] **Corbató, F.J., Merwin-Daggett, M. und Daley, R.C.** (1962): An Experimental Time-sharing System. Proceedings of the AIEE-IRE '62 Spring Joint Computer Conference, AIEE-IRE '62 (Spring), Seiten 335–344. ACM Press, New York, NY, USA. doi:10.1145/1460833.1460871. (Zitiert auf Seite 42.)

[CNV96] **Calin, T., Nicolaidis, M. und Velazco, R.** (1996): Upset hardened memory design for submicron CMOS technology. IEEE Transactions on Nuclear Science, 43(6): Seiten 2874–2878. ISSN 0018-9499. doi:10.1109/23.556880. (Zitiert auf Seite 6.)

[CNV04] **Correia, M., Neves, N.F. und Verissimo, P.** (2004): How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems. Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS '04), SRDS '04, Seiten 174–183. IEEE Computer Society Press, Washington, DC, USA. ISBN 0-7695-2239-4. (Zitiert auf Seite 54.)

[Con03] **Constantinescu, C.** (2003): Trends and challenges in VLSI circuit reliability. Micro, IEEE, 23(4): Seiten 14–19. ISSN 0272-1732. doi:10.1109/MM.2003.1225959. (Zitiert auf Seite 6.)

[CRA06] **Chang, J., Reis, G. und August, D.** (2006): Automatic Instruction-Level Software-Only Recovery. Proceedings of the 36th International Conference on Dependable Systems and Networks (DSN '06), Seiten 83–92. IEEE Computer Society Press, Washington, DC, USA. doi:10.1109/DSN.2006.15. (Zitiert auf den Seiten 38, 43 und 44.)

[DH12] **Döbel, B. und Härtig, H.** (2012): Who Watches the Watchmen? – Protecting Operating System Reliability Mechanisms. Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep '12). USENIX Association, Berkeley, CA, USA. (Zitiert auf Seite 42.)

[DHW09] **Dixit, A., Heald, R. und Wood, A.** (2009): Trends from ten years of soft error experimentation. Proceedings of the 5th Workshop on Silicon Errors in Logic – System Effects (SLSE '09). (Zitiert auf Seite 6.)

[DIN90] **DIN 40041** (1990): DIN 40041:1990-12: Zuverlässigkeit; Begriffe. Deutsches Institut für Normung. (Zitiert auf Seite 24.)

[DM02] **Duraes, J. und Madeira, H.** (2002): Emulation of software faults by educated mutations at machine-code level. Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE '03), Seiten 329–340. IEEE Computer Society Press, Washington, DC, USA. ISSN 1071-9458. doi:10.1109/ISSRE.2002.1173283. (Zitiert auf Seite 24.)

[Dre08] **Dremel, B.** (2008): Realisierung einer prototypischen Lagekontrolle für einen Modell-Quadrocopter. Studienarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[Dre11] **Dremel, B.** (2011): Ein erweiterter Software-Redundanzansatz für den Einsatz von Grafikkarten in sicherheitskritischen Anwendungen. Diplomarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[Ech90] **Echtle, K.** (1990): Fehlertoleranzverfahren. 1. Auflage. 322 Seiten, Springer, Berlin. ISBN 978-0-3875-2680-5. (Zitiert auf den Seiten 24, 27, 28, 29, 33, 40, 49, 51, 52, 53, 60, 77, 118 und 14.)

[EFB01] **Elrad, T., Filman, R.E. und Bader, A.** (2001): Aspect-Oriented Programming. Communications of the ACM, Seiten 29–32. (Zitiert auf Seite 42.)

[EJ09] **Ebert, C. und Jones, C.** (2009): Embedded Software: Facts, Figures, and Future. Computer, 42(4): Seiten 42–52. ISSN 0018-9162. doi:10.1109/MC.2009.118. (Zitiert auf Seite 9.)

[Fis09] **Fischer, R.** (2009): Elektrische Maschinen. 14. Auflage. 420 Seiten, Hanser, München. ISBN 9-7834-4641-7-540. (Zitiert auf den Seiten 109 und 5.)

[For89] **Forin, P.** (1989): Vital coded microprocessor principles and application for various transit systems. Proceedings of the IFAC IFIP/IFORS Symposium on Control, Computers, Communications in Transportation (CCCT '89), Seiten 79–84. (Zitiert auf den Seiten 7, 23, 24, 38, 39, 57, 63, 64, 78 und 83.)

[FPW97] **Franklin, G.F., Powell, J.D. und Workman, M.L.** (1997): Digital Control of Dynamic Systems. 3. Auflage. 850 Seiten, Addison-Wesley, Boston, MA, USA. ISBN 978-0-2018-2054-6. (Zitiert auf den Seiten 55, 71, 125, 127 und 129.)

[Fra09] **Franzmann, F.** (2009): Entwurf und Implementierung eines sicheren Kommunikationsframeworks für Flugdrohnen. Studienarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[Fra10] **Fraden, J.** (2010): Handbook of Modern Sensors: Physics, Designs, and Applications. 4. Auflage. 489 Seiten, Springer, New York, NY, USA. ISBN 978-1-4419-6465-6. (Zitiert auf den Seiten 59, 61, 107 und 3.)

[Fra11] **Franzmann, F.** (2011): Quality-aware Digital Signal Processing for Real-time Systems. Diplomarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[Fro77] **Frohwerk, R.A.** (1977): Signature analysis: A new digital field service method. Hewlett-Packard Journal, 28(9): Seiten 2–8. (Zitiert auf den Seiten 77 und 84.)

[FSS09] **Fetzer, C., Schiffel, U. und Süßkraut, M.** (2009): AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware. B. Buth, G. Rabe und T. Seyfarht (Hg.), Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '09), Seiten 283–296. Springer-Verlag, Heidelberg, Germany. ISBN 978-3-642-04467-0. doi:10.1007/978-3-642-04468-7_23. (Zitiert auf Seite 38.)

[GGR+04] **Gordon, M., Goldhagen, P., Rodbell, K., Zabel, T., Tang, H., Clem, J. und Bailey, P.** (2004): Measurement of the flux and energy spectrum of cosmic-ray induced neutrons on the ground. IEEE Transactions on Nuclear Science, 51(6): Seiten 3427–3434. ISSN 0018-9499. doi:10.1109/TNS.2004.839134. (Zitiert auf Seite 4.)

[Gol73] **Goldberg, R.P.** (1973): Architectural principles for virtual computer systems. Dissertation, Division of Engineering and Applied Sciences, Harvard University. (Zitiert auf Seite 8.)

[GRRV03] **Goloubeva, O., Rebaudengo, M., Reorda, M. und Violante, M.** (2003): Soft-error detection using control flow assertions. Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Seiten 581–588. IEEE Computer Society Press, Washington, DC, USA. ISSN 1550-5774. doi:10.1109/DFTVS.2003.1250158. (Zitiert auf Seite 32.)

[GRRV06a] **Goloubeva, O., Rebaudengo, M., Reorda, M.S. und Violante, M.** (2006): Software-Implemented Hardware Fault Tolerance. 1. Auflage. 228 Seiten, Springer-Verlag, New York, NY, USA. ISBN 0-387-26060-9. (Zitiert auf den Seiten 7, 24 und 41.)

[GRRV06b] **Goloubeva, O., Rebaudengo, M., Reorda, M.S. und Violante, M.** (2006): Software-Implemented Hardware Fault Tolerance. Springer-Verlag, Secaucus, NJ, USA. ISBN 0387260609. (Zitiert auf Seite 52.)

[GSVP03] **Gomaa, M., Scarbrough, C., Vijaykumar, T. und Pomeranz, I.** (2003): Transient-fault recovery for chip multiprocessors. Proceedings of the 30th International Symposium on Computer Architecture (ISCA '03), Seiten 98–109. IEEE Computer Society Press, Washington, DC, USA. ISSN 1063-6897. doi:10.1109/ISCA.2003.1206992. (Zitiert auf Seite 32.)

[Ham50] **Hamming, R.W.** (1950): Error detecting and error correcting codes. Bell System Technical Journal, 29(2): Seiten 147–160. (Zitiert auf den Seiten 6, 26, 36, 78 und 80.)

[Ham80] **Hamming, R.W.** (1980): Coding and information theory. Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN ISBN 0-13-139139-9. (Zitiert auf den Seiten 75 und 97.)

[Han01] **Hansen, P.** (2001): Classic Operating Systems : from Batch Processing to Distributed Systems. 1. Auflage. 597 Seiten, Springer, New York, NY, USA. ISBN 978-0-387-95113-3. (Zitiert auf Seite 1.)

[Har06] **Hardung, B.** (2006): Optimisation of the Allocation of Functions in Vehicle Networks. Shaker Verlag. ISBN 978-3-8322-5687-6. (Zitiert auf Seite 9.)

[HDL13] **Hoffmann, M., Dietrich, C. und Lohmann, D.** (2013): dOSEK: A Dependable RTOS for Automotive Applications. Proceedings of the 19th International Symposium on Dependable Computing (PRDC '13). IEEE Computer Society Press, Washington, DC, USA. Fast abstract. (Zitiert auf Seite 141.)

[Hei11] **Heigl, M.** (2011): Entwicklung und Konstruktion eines Leichtbau-Tragwerks für einen Quadrokopter. Bachelorarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[HG93] **Hennebert, C. und Guiho, G.** (1993): SACEM: A fault tolerant system for train speed control. Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing (FTCS '93), Seiten 624–628. IEEE Computer Society Press, Washington, DC, USA. ISSN 0731-3071. doi:10.1109/FTCS.1993.627365. (Zitiert auf Seite 39.)

[HG13] **Hafner, T. und Gaal, W.** (2013): A Long-term Mutually Beneficial Cooperation. ATZextra worldwide, 18(9): Seiten 60–62. ISSN 2195-1470. doi:10.1007/s40111-013-0020-4. (Zitiert auf den Seiten 109 und 5.)

[HHJ90] **Horst, R.W., Harris, R.L. und Jardine, R.L.** (1990): Multiple Instruction Issue in the NonStop Cyclone Processor. Proceedings of the 17th International Symposium on Computer Architecture (ISCA '90), ISCA '90, Seiten 216–226. ACM Press, New York, NY, USA. ISBN 0-89791-366-3. doi:10.1145/325164.325147. (Zitiert auf Seite 32.)

[Hig07] **HighTec EDV-Systeme GmbH** (2007): EasyRun TC1796 – Handbuch. HighTec EDV-Systeme GmbH, Feldmannstr. 98, D-66119 Saarbrücken. (Zitiert auf den Seiten 109 und 5.)

[HK08] **Harl, S. und Kotulla, S.** (2008): Realisierung einer prototypischen Steuerungssoftware für einen Quadcopter. Studienarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[HS80] **Hirai, K. und Satoh, Y.** (1980): Stability of a system with variable time delay. IEEE Transactions on Automatic Control, 25(3): Seiten 552–554. ISSN 0018-9286. doi:10.1109/TAC.1980.1102355. (Zitiert auf Seite 125.)

[IEC98] **IEC 61508-3** (1998): IEC 61508-3: - Functional safety of electrical/electronic/-programmable electronic safety-related systems – Part 3: Software requirements. International Electrotechnical Commission, Geneva, Switzerland. (Zitiert auf Seite 141.)

[Inf07] **Infineon Technologies AG** (2007): TC1796 – Benutzerhandbuch (V2.0). Infineon Technologies AG, St.-Martin-Str. 53, D-81669 München. (Zitiert auf den Seiten 109, 115, 5 und 11.)

[ISO11a] **ISO 26262-5** (2011): ISO 26262-5:2011: Road vehicles – Functional safety – Part 5: Product development at the hardware level. International Organization for Standardization, Geneva, Switzerland. (Zitiert auf Seite 9.)

[ISO11b] **ISO 26262-8** (2011): ISO 26262-9:2011: Road vehicles – Functional safety – Part 8: Supporting processes. International Organization for Standardization, Geneva, Switzerland. (Zitiert auf Seite 141.)

[ITY+10] **Ibe, E., Taniguchi, H., Yahagi, Y., Shimbo, K.i. und Toba, T.** (2010): Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. IEEE Transactions on Electron Devices, 57(7): Seiten 1527–1538. ISSN 0018-9383. doi:10.1109/TED.2010.2047907. (Zitiert auf Seite 6.)

[JED06] **JEDEC** (2006): Standard: Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices (JESD89A). JEDEC Solid State Technology Association, Arlington, VA. (Zitiert auf Seite 4.)

[JLT85] **Jensen, E.D., Locke, C.D. und Tokuda, H.** (1985): A Time-Driven Scheduling Model for Real-Time Operating Systems. Proceedings of the 6th Real-Time Systems Symposium (RTSS '85), Seiten 112–122. IEEE Computer Society Press, Los Alamitos, CA, USA. ISBN 0-8186-0675-4.

[KDK+89] **Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C. und Zainlinger, R.** (1989): Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. IEEE Micro, 9(1): Seiten 25–40. ISSN 0272-1732. doi:10.1109/40.16792. (Zitiert auf den Seiten 40 und 126.)

[KG02] **Kuo, B.C. und Golnaraghi, F.** (2002): Automatic Control Systems. 8. Auflage. 624 Seiten, Wiley-Blackwell, New York, NY, USA. ISBN 978-0-4711-3476-3. (Zitiert auf den Seiten 55, 125, 126 und 130.)

[KH04] **Karnik, T. und Hazucha, P.** (2004): Characterization of soft errors caused by single event upsets in CMOS processes. IEEE Transactions on Dependable and Secure Computing, 1(2): Seiten 128–143. ISSN 1545-5971. doi:10.1109/TDSC.2004.14. (Zitiert auf Seite 5.)

[KIR+99] **Kalbarczyk, Z., Iyer, R., Ries, G., Patel, J., Lee, M. und Xiao, Y.** (1999): Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. IEEE Transactions on Software Engineering, 25(5): Seiten 619–632. ISSN 0098-5589. doi:10.1109/32.815322. (Zitiert auf Seite 99.)

[KKA93] **Kanawati, G.A., Kanawati, N.A. und Abraham, J.A.** (1993): EMAX: An automatic extractor of high-level error models. Computing Aerospace Conference, Seiten 1297ï¿œ–1306. (Zitiert auf Seite 99.)

[KKA95] **Kanawati, G.A., Kanawati, N.A. und Abraham, J.A.** (1995): FERRARI: A Flexible Software-Based Fault and Error Injection System. IEEE Transactions on Computers, 44: Seiten 248–260. ISSN 0018-9340. (Zitiert auf den Seiten 76 und 89.)

[Kla12] **Klaus, T.** (2012): Semi-autonome Höhenregelung eines Quattrokopters unter Einsatz eines heterogenen Sensorsystems. Projektarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[KLR94] **Klein, M., Lehoczky, J. und Rajkumar, R.** (1994): Rate-monotonic analysis for real-time industrial computing. IEEE Computer, 27(1): Seiten 24–33. ISSN 0018-9162. doi:10.1109/2.248876. (Zitiert auf Seite 128.)

[Kop97] **Kopetz, H.** (1997): Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers. ISBN 0-7923-9894-7. (Zitiert auf den Seiten 40, 53 und 126.)

[KVV+02] **Karnik, T., Vangal, S., Veeramachaneni, V., Hazucha, P., Erraguntla, V. und Borkar, S.** (2002): Selective Node Engineering for Chip-level Soft Error Rate Improvement [in CMOS]. Proceedings of the Symposium on VLSI Circuits, Seiten 204–205. IEEE Press, New York, NY, USA. doi:10.1109/VLSIC.2002.1015084. (Zitiert auf Seite 6.)

[LA04] **Lattner, C. und Adve, V.** (2004): LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). IEEE Computer Society Press, Washington, DC, USA. (Zitiert auf Seite 38.)

[LABK90] **Laprie, J.C., Arlat, J., Beounes, C. und Kanoun, K.** (1990): Definition and analysis of hardware- and software-fault-tolerant architectures. Computer, 23(7): Seiten 39–51. ISSN 0018-9162. doi:10.1109/2.56851. (Zitiert auf den Seiten 8 und 21.)

[Lal00] **Lala, P.K.** (2000): Self-checking and fault-tolerant digital design. The Morgan Kaufmann Series in Computer Architecture and Design, 1. Auflage. 400 Seiten, Morgan Kaufmann. ISBN 978-0-1243-4370-2. (Zitiert auf Seite 50.)

[Law96] **Lawton, K.P.** (1996): Bochs: A Portable PC Emulator for Unix/X. Linux Journal, 1996(29es): Seite 7. (Zitiert auf Seite 89.)

[LCE89] **Lorczak, P., Caglayan, A. und Eckhardt, D.** (1989): A theoretical investigation of generalized voters for redundant systems. Digest of papers, The 19th International Symposium on Fault-Tolerant Computing (FTCS '89), Seiten 444–451. doi:10.1109/FTCS.1989.105617. (Zitiert auf Seite 60.)

[LHSP+09] **Lohmann, D., Hofer, W., Schröder-Preikschat, W., Streicher, J. und Spinczyk, O.** (2009): CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems. Proceedings of the 2009 USENIX Annual Technical Conference, Seiten 215–228. USENIX Association, Berkeley, CA, USA. ISBN 978-1-931971-68-3. (Zitiert auf den Seiten 110 und 6.)

[Lim11] **Limmer, F.** (2011): Entwicklung und Konstruktion eines modularen, robusten Tragwerks für einen Quadrokopter. Bachelorarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[Liu00] **Liu, J.W.S.** (2000): Real-Time Systems. Prentice Hall PTR, Englewood Cliffs, NJ, USA. ISBN 0-13-099651-3. (Zitiert auf den Seiten 71 und 72.)

[LM01] **Liggesmeyer, P. und Maeckel, O.** (2001): Quantifying the Reliability of Embedded Systems by Automated Analysis. Proceedings of the 31st International Conference on Dependable Systems and Networks (DSN '01), Seiten 89–96. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-1101-5. (Zitiert auf Seite 32.)

[LRC05] **Lucchetti, D., Reinhardt, S.K. und Chen, P.M.** (2005): ExtraVirt: detecting and recovering from transient processor faults. Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05), Seiten 1–8. ACM Press, New York, NY, USA. ISBN 1-59593-079-5. doi:10.1145/1095810.1118621. (Zitiert auf Seite 41.)

[LSHC07] **Li, X., Shen, K., Huang, M.C. und Chu, L.** (2007): A Memory Soft Error Measurement on Production Systems. Proceedings of the 2007 USENIX Annual Technical Conference, Seiten 1–14. USENIX Association, Berkeley, CA, USA. ISBN 999-8888-77-6. (Zitiert auf den Seiten 76 und 87.)

[LSP82] **Lamport, L., Shostak, R. und Pease, M.** (1982): The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems, 4(3): Seiten 382–401. ISSN 0164-0925. doi:10.1145/357172.357176. (Zitiert auf Seite 53.)

[Lue64] **Luenberger, D.** (1964): Observing the State of a Linear System. IEEE Transactions on Military Electronics, 8(2): Seiten 74–80. ISSN 0536-1559. doi:10.1109/TME.1964.4323124. (Zitiert auf Seite 132.)

[LV62] **Lyons, R. und Vanderkulk, W.** (1962): The Use of Triple-Modular Redundancy to Improve Computer Reliability. IBM Journal of Research and Development, 6(2): Seiten 200–209. ISSN 0018-8646. doi:10.1147/rd.62.0200. (Zitiert auf Seite 49.)

[Man67] **Mandelbaum, D.** (1967): Arithmetic codes with large distance. IEEE Transactions on Information Theory, 13(2): Seiten 237–242. ISSN 0018-9448. doi:10.1109/TIT.1967.1054015. (Zitiert auf den Seiten 39 und 98.)

[Mas64] **Massey, J.L.** (1964): Survey of residue coding for arithmetic errors. International Computation Center Bulletin, 3(4): Seiten 3–17. (Zitiert auf Seite 98.)

[MER05] **Mukherjee, S., Emer, J. und Reinhardt, S.** (2005): The Soft Error Problem: An Architectural Perspective. Proceedings of the 11th IEEE International Symposium on High-Performance Computer Architecture, Seiten 243–247. IEEE Computer Society Press, Washington, DC, USA. ISSN 1530-0897. doi:10.1109/HPCA.2005.37. (Zitiert auf den Seiten 22 und 53.)

[MGO72] **Massey, J.L., García, O.N. und Oscar, N.** (1972): Error-Correcting Codes in Computer Arithmetic. J.T. Tou (Hg.), Advances in Information Systems Science, Seiten 273–326. Springer. ISBN 978-1-4615-9055-2. doi:10.1007/978-1-4615-9053-8_5. (Zitiert auf Seite 39.)

[MHZA03] **Maiz, J., Hareland, S., Zhang, K. und Armstrong, P.** (2003): Characterization of multi-bit soft error events in advanced SRAMs. Proceedings of the IEEE International Electron Devices Meeting (IEDM '03), Seiten 21.4.1–21.4.4. IEEE Press, New York, NY, USA. doi:10.1109/IEDM.2003.1269335. (Zitiert auf den Seiten 76, 81 und 87.)

[MKR02] **Mukherjee, S., Kontz, M. und Reinhardt, S.** (2002): Detailed design and evaluation of redundant multi-threading alternatives. Proceedings of the 29th International Symposium on Computer Architecture (ISCA '02), Seiten 99–110. IEEE Computer Society Press, Washington, DC, USA. ISSN 1063-6897. doi:10.1109/ISCA.2002.1003566. (Zitiert auf Seite 32.)

[MKT+11] **Maniatakos, M., Karimi, N., Tirumurti, C., Jas, A. und Makris, Y.** (2011): Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller. IEEE Transactions on Computers, 60(9): Seiten 1260–1273. ISSN 0018-9340. doi:10.1109/TC.2010.60. (Zitiert auf Seite 99.)

[MM88] **Mahmood, A. und McCluskey, E.J.** (1988): Concurrent Error Detection Using Watchdog Processors-A Survey. IEEE Transactions on Computers, 37: Seiten 160–174. ISSN 0018-9340. (Zitiert auf Seite 32.)

[MS09] **Medwed, M. und Schmidt, J.M.** (2009): Coding Schemes for Arithmetic and Logic Operations - How Robust Are They? H. Youm und M. Yung (Hg.), Information Security Applications, Band 5932 von *Lecture Notes in Computer Science*, Seiten 51–65. Springer, Heidelberg. ISBN 978-3-642-10837-2. doi:10.1007/978-3-642-10838-9_5. (Zitiert auf Seite 97.)

[Muk08] **Mukherjee, S.** (2008): Architecture Design for Soft Errors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 978-0-12-369529-1. (Zitiert auf den Seiten 8, 10, 21, 22, 26, 32, 41, 43, 52 und 80.)

[MW79] **May, T. und Woods, M.** (1979): Alpha-particle-induced soft errors in dynamic memories. IEEE Transactions on Electron Devices, 26(1): Seiten 2–9. ISSN 0018-9383. doi:10.1109/T-ED.1979.19370. (Zitiert auf den Seiten 3 und 76.)

[MWE+03] **Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K. und Austin, T.** (2003): A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Seite 29. IEEE Computer Society Press, Washington, DC, USA. ISBN 0-7695-2043-X. (Zitiert auf Seite 53.)

[NDO11] **Nightingale, E.B., Douceur, J.R. und Orgovan, V.** (2011): Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. C.M. Kirsch und G. Heiser (Hg.), Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11), Seiten 343–356. ACM Press, New York, NY, USA. ISBN 978-1-4503-0634-8. doi:10.1145/1966445.1966477. (Zitiert auf den Seiten 6, 10, 53, 76 und 87.)

[Nic03] **Nicolaidis, M.** (2003): Carry checking/parity prediction adders and ALUs. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 11(1): Seiten 121–128. ISSN 1063-8210. doi:10.1109/TVLSI.2002.800526. (Zitiert auf Seite 7.)

[NN99] **Noufal, I. und Nicolaidis, M.** (1999): A CAD framework for generating self-checking multipliers based on residue codes. Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '99), Seiten 122–129. IEEE Computer Society Press, Washington, DC, USA. doi:10.1109/DATE.1999.761107. (Zitiert auf Seite 37.)

[Nor96] **Normand, E.** (1996): Single event upset at ground level. IEEE Transactions on Nuclear Science, 43(6): Seiten 2742–2750. ISSN 0018-9499. doi:10.1109/23.556861. (Zitiert auf Seite 3.)

[NSV04] **Nicolescu, B., Savaria, Y. und Velazco, R.** (2004): Software detection mechanisms providing full coverage against single bit-flip faults. IEEE Transactions on Nuclear Science, 51(6): Seiten 3510–3518. ISSN 0018-9499. doi:10.1109/TNS.2004.839110. (Zitiert auf Seite 44.)

[OMM02] **Oh, N., Mitra, S. und McCluskey, E.** (2002): ED4I: Error Detection by Diverse Data and Duplicated Instructions. IEEE Transactions on Computers, 51(2): Seiten 180–199. ISSN 0018-9340. doi:10.1109/12.980007. (Zitiert auf Seite 38.)

[OR95] **Ohlsson, J. und Rimen, M.** (1995): Implicit signature checking. Digest of Papers, The 25th International Symposium on Fault-Tolerant Computing (FTCS '95), Seiten 218–227. IEEE Press, New York, NY, USA. doi:10.1109/FTCS.1995.466976. (Zitiert auf Seite 32.)

[OSM02a] **Oh, N., Shirvani, P. und McCluskey, E.** (2002): Control-flow checking by software signatures. IEEE Transactions on Reliability, 51(1): Seiten 111–122. ISSN 0018-9529. doi:10.1109/24.994926. (Zitiert auf Seite 32.)

[OSM02b] **Oh, N., Shirvani, P. und McCluskey, E.** (2002): Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability, 51(1): Seiten 63–75. ISSN 0018-9529. doi:10.1109/24.994913. (Zitiert auf Seite 43.)

[Oze92] **Ozello, P.** (1992): The coded microprocessor certification. H.H. Frey (Hg.), Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP '92), Seiten 185–190. Pergamon Press, Oxford, UK. ISBN 978-0-0804-1893-3. (Zitiert auf Seite 78.)

[PBdST05] **Potop-Butucaru, D., de Simone, R. und Talpin, J.P.** (2005): Embedded Systems Handbook, Kapitel 8: The Synchronous hypothesis and Synchronous Languages, Seiten 8–1–8–23. CRC Press. ISBN 978-0-8493-2824-4. doi:10.1201/9781420038163.ch8. (Zitiert auf den Seiten 10 und 125.)

[PMC06] **Pounds, P., Mahony, R. und Corke, P.** (2006): Modelling and Control of a Quad-Rotor Robot. Proceedings of the 2006 Australasian Conference on Robotics & Automation, Seiten 1–10. Australian Robotics and Automation Association Inc., Auckland, New Zealand. (Zitiert auf den Seiten 107 und 3.)

[Pol94] **Poledna, S.** (1994): Replica determinism in distributed real-time systems: a brief survey. Real-Time Systems Journal, 6(3): Seiten 289–316. ISSN 0922-6443. doi:10.1007/BF01088629. (Zitiert auf Seite 40.)

[Pra96] **Pradhan, D.K.** (1996): Fault-Tolerant Computer System Design. 1. Auflage. 550 Seiten, Prentice-Hall, Upper Saddle River, NJ, USA. ISBN 0-13-057887-8. (Zitiert auf Seite 24.)

[Pul01] **Pullum, L.L.** (2001): Software fault tolerance techniques and implementation. Artech House, Inc., Norwood, MA, USA. ISBN 1-58053-137-7. (Zitiert auf Seite 60.)

[PW72] **Peterson, W.W. und Weldon, E.J.** (1972): Error-correcting codes. 2. Auflage. 572 Seiten, MIT Press, Cambridge, MA, USA. ISBN 978-0-2621-6039-1. (Zitiert auf den Seiten 36, 97 und 98.)

[Rao70] **Rao, T.R.N.** (1970): Biresidue Error-Correcting Codes for Computer Arithmetic. IEEE Transactions on Computers, C-19(5): Seiten 398–402. ISSN 0018-9340. doi: 10.1109/T-C.1970.222937. (Zitiert auf Seite 39.)

[Rao74] **Rao, T.R.N.** (1974): Error Coding for Arithmetic Processors. 1. Auflage. 218 Seiten, Academic Press, Orlando, FL, USA. ISBN 978-0-1258-0750-0. (Zitiert auf den Seiten 34, 36 und 97.)

[Ray94] **Ray, A.** (1994): Output feedback control under randomly varying distributed delays. Guidance, Control, and Dynamics, 17(4): Seiten 701–711. (Zitiert auf Seite 125.)

[RCBS07] **Rao, R., Chopra, K., Blaauw, D. und Sylvester, D.** (2007): Computing the Soft Error Rate of a Combinational Logic Circuit Using Parameterized Descriptors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(3): Seiten 468–479. ISSN 0278-0070. doi:10.1109/TCAD.2007.891036. (Zitiert auf Seite 53.)

[RCV+05a] **Reis, G., Chang, J., Vachharajani, N., Mukherjee, S., Rangan, R. und August, D.** (2005): Design and evaluation of hybrid fault-detection systems. Proceedings of the 32nd International Symposium on Computer Architecture (ISCA '05), Seiten 148–159. IEEE Computer Society Press, Washington, DC, USA. ISSN 1063-6897. doi:10.1109/ISCA.2005.21. (Zitiert auf den Seiten 43 und 44.)

[RCV+05b] **Reis, G., Chang, J., Vachharajani, N., Rangan, R., August, D. und Mukherjee, S.** (2005): Software-controlled fault tolerance. ACM Transactions on Architecture and Code Optimization (TACO '05), 2(4): Seiten 366–396. ISSN 1544-3566. doi: 10.1145/1113841.1113843. (Zitiert auf Seite 43.)

[Reb12] **Rebhan, A.** (2012): Entwicklung einer zustandsbasierten Fluglageregelung für einen Quadrokopter. Bachelorarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[RG05] **Rosenblum, M. und Garfinkel, T.** (2005): Virtual machine monitors: current technology and future trends. Computer, 38(5): Seiten 39–47. ISSN 0018-9162. doi: 10.1109/MC.2005.176. (Zitiert auf Seite 41.)

[RHKSP06] **Reiser, H.P., Hauck, F.J., Kapitza, R. und Schröder-Preikschat, W.** (2006): Hypervisor-Based Redundant Execution on a Single Physical Host. Proceedings of the 6th European Dependable Computing Conference (EDCC '06)– Supplemental Volume, Seiten 67–68. (Zitiert auf Seite 41.)

[RM00] **Reinhardt, S.K. und Mukherjee, S.S.** (2000): Transient Fault Detection via Simultaneous Multithreading. Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00), Seiten 25–36. ACM Press, New York, NY, USA. ISBN 1-58113-232-8. doi:10.1145/339647.339652. (Zitiert auf den Seiten 27, 32, 40 und 42.)

[RR99] **Rebaudengo, M. und Reorda, M.S.** (1999): Evaluating the Fault Tolerance Capabilities of Embedded Systems via BDM. Proceedings of the 17TH IEEE VLSI Test Symposium (VTS '99), Seiten 452–457. IEEE Computer Society Press, Washington, DC, USA. ISBN 0-7695-0146-X. doi:10.1109/VTEST.1999.766703. (Zitiert auf den Seiten 115 und 11.)

[RRTV99] **Rebaudengo, M., Reorda, M., Torchiano, M. und Violante, M.** (1999): Soft-error detection through software fault-tolerance techniques. Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '99), Seiten 210–218. IEEE Computer Society Press, Washington, DC, USA. ISSN 1550-5774. doi:10.1109/DFTVS.1999.802887. (Zitiert auf Seite 44.)

[RRVT01] **Rebaudengo, M., Reorda, M., Violante, M. und Torchiano, M.** (2001): A source-to-source compiler for generating dependable software. Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation, Seiten 33–42. IEEE Computer Society Press, Washington, DC, USA. doi:10.1109/SCAM.2001.972664. (Zitiert auf Seite 44.)

[SAC+99] **Slegel, T., Averill, R.M., I., Check, M., Giamei, B., Krumm, B., Krygowski, C., Li, W., Liptay, J., MacDougall, J., McPherson, T., Navarro, J., Schwarz, E., Shum, K. und Webb, C.** (1999): IBM's S/390 G5 microprocessor design. IEEE Micro, 19(2): Seiten 12–23. ISSN 0272-1732. doi:10.1109/40.755464. (Zitiert auf Seite 32.)

[SBM+09] **Shye, A., Blomstedt, J., Moseley, T., Reddi, V.J. und Connors, D.A.** (2009): PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures. IEEE Transactions on Dependable and Secure Computing, 6: Seiten 135–148. ISSN 1545-5971. doi:10.1109/TDSC.2008.62. (Zitiert auf Seite 42.)

[SC95] **Shin, K. und Cui, X.** (1995): Computing time delay and its effects on real-time control systems. IEEE Transactions on Control Systems Technology, 3(2): Seiten 218–224. ISSN 1063-6536. doi:10.1109/87.388130. (Zitiert auf den Seiten 123, 125 und 126.)

[Sch90] **Schneider, F.B.** (1990): Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. ACM Computing Surveys, 22(4): Seiten 299–319. ISSN 0360-0300. doi:10.1145/98163.98167. (Zitiert auf Seite 52.)

[Sch10] **Schedel, J.** (2010): Entwurf und Implementierung einer Systemabstraktionsschicht für das I4Copter-Framework. Diplomarbeit, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg. (Zitiert auf den Seiten 107 und 3.)

[Sch11a] **Scheler, F.** (2011): Atomic Basic Blocks. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, Technische Fakultät. (Zitiert auf Seite 141.)

[Sch11b] **Schiffel, U.** (2011): Hardware Error Detection Using AN-Codes. Dissertation, Technische Universität Dresden, Fakultät Informatik. (Zitiert auf den Seiten 32, 36, 37, 38, 39, 40, 64, 65, 78, 81, 82, 84 und 100.)

[SH04] **Segura, J. und Hawkins, C.F.** (2004): CMOS Electronics: How It Works, How It Fails. 1. Auflage. 348 Seiten, John Wiley & Sons, Hoboken, NJ, USA. ISBN 978-0471476696. (Zitiert auf den Seiten 2 und 3.)

[Sha49] **Shannon, C.** (1949): Communication in the Presence of Noise. Proceedings of the IRE, 37(1): Seiten 10–21. ISSN 0096-8390. doi:10.1109/JRPROC.1949.232969. (Zitiert auf den Seiten 6 und 36.)

[SHK+12] **Schirmeier, H., Hoffmann, M., Kapitza, R., Lohmann, D. und Spinczyk, O.** (2012): FAIL*: Towards a Versatile Fault-Injection Experiment Framework. G. Mühl, J. Richling und A. Herkersdorf (Hg.), 25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings, Band 200 von *Lecture Notes in Informatics*, Seiten 201–210. Gesellschaft für Informatik. ISBN 978-3-88579-294-9. (Zitiert auf Seite 89.)

[SKK+02] **Shivakumar, P., Kistler, M., Keckler, S.W., Burger, D. und Alvisi, L.** (2002): Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN '02), Seiten 389–398. IEEE Computer Society Press, Washington, DC, USA. doi:10.1109/DSN.2002.1028924. (Zitiert auf Seite 5.)

[SKK+08] **Sanda, P., Kellington, J., Kudva, P., Kalla, R., McBeth, R.B., Ackaret, J., Lockwood, R., Schumann, J. und Jones, C.R.** (2008): Soft-error resilience of the IBM POWER6 processor. IBM Journal of Research and Development, 52(3): Seiten 275–284. ISSN 0018-8646. doi:10.1147/rd.523.0275. (Zitiert auf Seite 37.)

[SMM10] **Steindl, M., Mottok, J. und Meier, H.** (2010): SES-based framework for fault-tolerant systems. Proceedings of the 8th Workshop on Intelligent Solutions in Embedded Systems (WISES '10), Seiten 12–16. doi:10.1109/WISES.2010.5548427. (Zitiert auf Seite 97.)

[SMR+07] **Shye, A., Moseley, T., Reddi, V.J., Blomstedt, J. und Connors, D.A.** (2007): Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN '07), Seiten 297–306. IEEE Computer Society Press, Washington, DC, USA. ISBN 0-7695-2855-4. doi:10.1109/DSN.2007.98. (Zitiert auf den Seiten 8, 12 und 42.)

[SS87] **Schuette, M.A. und Shen, J.P.** (1987): Processor Control Flow Monitoring Using Signatured Instruction Streams. IEEE Transactions on Computers, 36: Seiten 264–276. ISSN 0018-9340. (Zitiert auf den Seiten 32 und 43.)

[SS98] **Siewiorek, D.P. und Swarz, R.S.** (1998): Reliable computer systems: design and evaluation. 3. Auflage. 908 Seiten, A.K. Peters, Wellesley, MA, USA. ISBN 978-1-5688-1092-8. (Zitiert auf Seite 7.)

[SSK+06] **Seifert, N., Slankard, P., Kirsch, M., Narasimham, B., Zia, V., Brookreson, C., Vo, A., Mitra, S., Gill, B. und Maiz, J.** (2006): Radiation-Induced Soft Error Rates of Advanced CMOS Bulk Devices. Proceedings of the 44th IEEE International Symposium on Reliability Physics, Seiten 217–225. IEEE Press, New York, NY, USA. doi:10.1109/RELPHY.2006.251220. (Zitiert auf Seite 6.)

[SSP11] **Scheler, F. und Schröder-Preikschat, W.** (2011): The Real-Time Systems Compiler: migrating event-triggered systems to time-triggered systems. Software: Practice and Experience, 41(12): Seiten 1491–1515. ISSN 1097-024X. doi:10.1002/spe.1099. (Zitiert auf Seite 141.)

[SSSF10] **Schiffel, U., Schmitt, A., Süßkraut, M. und Fetzer, C.** (2010): ANB- and ANBDmem-encoding: detecting hardware errors in software. E. Schoitsch (Hg.), Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '10), Seiten 169–182. Springer-Verlag, Heidelberg, Germany. ISBN 978-3-642-15650-2. doi:10.1007/978-3-642-15651-9_13. (Zitiert auf den Seiten 32, 39, 65 und 100.)

[Sta02] **Stathis, J.H.** (2002): Reliability limits for the gate insulator in CMOS technology. IBM Journal of Research and Development, 46(2.3): Seiten 265–286. ISSN 0018-8646. doi:10.1147/rd.462.0265. (Zitiert auf Seite 2.)

[Tan06] **Tanenbaum, A.S.** (2006): Structured Computer Organization. 5. Auflage. 777 Seiten, Prentice-Hall, Upper Saddle River, NJ, USA. ISBN 978-0-1314-8521-1. (Zitiert auf Seite 1.)

[TB82] **Taylor, D. und Black, J.** (1982): Principles of Data Structure Error Correction. IEEE Transactions on Computers, C-31(7): Seiten 602–608. ISSN 0018-9340. doi:10.1109/TC.1982.1676057. (Zitiert auf Seite 33.)

[Tew02] **Tewari, A.** (2002): Modern control design with MATLAB and SIMULINK. 1. Auflage. 520 Seiten, John Wiley & Sons, New York, NY, USA. ISBN 978-0-4714-9679-3. (Zitiert auf den Seiten 55, 127 und 128.)

[TLM+09] **Tremblay, G., Leveille, P., Mccollum, J., Pratt, M.J. und Bissett, T.D.** (2009). Schutzrecht EP1496434: Fault Resilient/Fault Tolerant Computing. (Erteilt: 23. Dezember 2009). (Zitiert auf den Seiten 8 und 41.)

[TN93] **Taber, A. und Normand, E.** (1993): Single event upset in avionics. IEEE Transactions on Nuclear Science, 40(2): Seiten 120–126. ISSN 0018-9499. doi:10.1109/23.212327. (Zitiert auf den Seiten 3 und 5.)

[Tör98] **Törngren, M.** (1998): Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems. Real-Time Systems Journal, 14: Seiten 219–250. ISSN 0922-6443. doi:10.1023/A:1007964222989. (Zitiert auf den Seiten 123, 125, 126 und 127.)

[TS86] **Taylor, D. und Seger, C.J.** (1986): Robust Storage Structures for Crash Recovery. IEEE Transactions on Computers, C-35(4): Seiten 288–295. ISSN 0018-9340. doi:10.1109/TC.1986.1676761. (Zitiert auf Seite 33.)

[TSK+11] **Thomm, I., Stilkerich, M., Kapitza, R., Lohmann, D. und Schröder-Preikschat, W.** (2011): Automated Application of Fault Tolerance Mechanisms in a Component-Based System. JTRES '11: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems, Seiten 87–95. ACM Press, New York, NY, USA. ISBN 978-1-4503-0731-4. doi:10.1145/2043910.2043925. (Zitiert auf Seite 141.)

[UBD02] **Unsöld, A., Baschek, B. und Duschl, W.J.** (2002): Der neue Kosmos: Einführung in die Astronomie und Astrophysik. 7. Auflage. 577 Seiten, Springer Spektrum, Heidelberg. ISBN 978-3-5404-2177-1. (Zitiert auf Seite 4.)

[UFH+12] **Ulbrich, P., Franzmann, F., Harkort, C., Hoffmann, M., Klaus, T., Rebhan, A. und Schröder-Preikschat, W.** (2012): Taking Control: Modular and Adaptive Robotics Process Control Systems. P. Payeur und P. Ben-Tzvi (Hg.), Proceedings of the 10th IEEE International Symposium on Robotic and Sensors Environments (ROSE '12), Seiten 55–60. IEEE Computer Society Press, Washington, DC, USA. ISBN 978-1-4673-2705-3. doi:10.1109/ROSE.2012.6402632. (Zitiert auf den Seiten 105, 106, 107, 132, 1, 2 und 3.)

[UHK+12] **Ulbrich, P., Hoffmann, M., Kapitza, R., Lohmann, D., Schröder-Preikschat, W. und Schmid, R.** (2012): Eliminating Single Points of Failure in Software-Based Redundancy. Proceedings of the 9th European Dependable Computing Conference (EDCC '12), Seiten 49–60. IEEE Computer Society Press, Washington, DC, USA. ISBN 978-1-4673-0938-7. doi:10.1109/EDCC.2012.21. (Zitiert auf den Seiten 107 und 3.)

[UKH+11] **Ulbrich, P., Kapitza, R., Harkort, C., Schmid, R. und Schröder-Preikschat, W.** (2011): I4Copter: An Adaptable and Modular Quadrotor Platform. Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11), Seiten 380–396. ACM Press, New York, NY, USA. ISBN 978-1-4503-0113-8. (Zitiert auf den Seiten 105, 107, 1 und 3.)

[VHM03] **Venkatasubramanian, R., Hayes, J. und Murray, B.** (2003): Low-cost on-line fault detection using control flow assertions. Proceedings of the 9th IEEE On-Line Testing Symposium (IOLTS '03)., Seiten 137–143. Washington, DC, USA. doi: 10.1109/OLT.2003.1214380. (Zitiert auf Seite 32.)

[VMw06] **VMware, Inc.** (2006). Virtualization Overview. Weißbuch. (Zitiert auf Seite 8.)

[VN56] **Von Neumann, J.** (1956): Probabilistic logics and the synthesis of reliable organisms from unreliable components. Automata studies, 34: Seiten 43–98. (Zitiert auf den Seiten 6, 54 und 55.)

[VPC02] **Vijaykumar, T.N., Pomeranz, I. und Cheng, K.** (2002): Transient-fault Recovery Using Simultaneous Multithreading. Proceedings of the 29th International Symposium on Computer Architecture (ISCA '02), ISCA '02, Seiten 87–98. IEEE Computer Society Press, Washington, DC, USA. ISBN 0-7695-1605-X. doi: 10.1145/545214.545226. (Zitiert auf Seite 32.)

[WF07] **Wappler, U. und Fetzer, C.** (2007): Software Encoded Processing: Building Dependable Systems with Commodity Hardware. F. Saglietti und N. Oster (Hg.), Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '07), Seiten 356–369. Springer-Verlag, Heidelberg, Germany. ISBN 978-3-540-75100-7. doi:10.1007/978-3-540-75101-4_34. (Zitiert auf den Seiten 8, 39 und 100.)

[Wil02] **Wilkins, D.** (2002): The bathtub curve and product failure behavior. Reliability HotWire, 21: Seite 17. (Zitiert auf Seite 2.)

[WKWY07] **Wang, C., Kim, H., Wu, Y. und Ying, V.** (2007): Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. Proceedings of the International Symposium on Code Generation and Optimization (CGO '07), Seiten 244–258. Washington, DC, USA. doi:10.1109/CGO.2007.7. (Zitiert auf Seite 42.)

[WNT95] **Wittenmark, B., Nilsson, J. und Törngren, M.** (1995): Timing problems in real-time control systems. Proceedings of the American Control Conference, Band 3, Seiten 2000–2004. IEEE Press, New York, NY, USA. ISBN 0-7803-2445-5. doi:10.1109/ACC.1995.531240. (Zitiert auf Seite 125.)

[WP05] **Wang, N.J. und Patel, S.J.** (2005): ReStore: Symptom Based Soft Error Detection in Microprocessors. Proceedings of the 35th International Conference on Dependable Systems and Networks (DSN '05), Seiten 30–39. IEEE Computer Society Press, Washington, DC, USA. ISBN 0-7695-2282-3. doi:10.1109/DSN.2005.82. (Zitiert auf Seite 32.)

[WQRP04] **Wang, N., Quek, J., Rafacz, T. und Patel, S.** (2004): Characterizing the effects of transient faults on a high-performance processor pipeline. Proceedings of the International Conference on Dependable Systems and Networks (DSN '04), Seiten 61–70. IEEE Computer Society Press, Washington, DC, USA. doi:10.1109/DSN.2004.1311877. (Zitiert auf Seite 53.)

[Yeh96] **Yeh, Y.** (1996): Triple-triple redundant 777 primary flight computer. Proceedings of the 1996 IEEE Aerospace Applications Conference, Seiten 293–307. IEEE Computer Society Press, Washington, DC, USA. ISBN 978-0780331969. doi:10.1109/AERO.1996.495891. (Zitiert auf den Seiten 5, 25, 32 und 123.)

[ZCM+96] **Ziegler, J., Curtis, H.W., Muhlfeld, H., Montrose, C., Chin, B., Nicewicz, M., Russell, C.A., Wang, W.Y., Freeman, L.B., Hosier, P., LaFave, L.E., Walsh, J., Orro, J.M., Unger, G.J., Ross, J.M., O'Gorman, T., Messina, B., Sullivan, T., Sykes, A.J., Yourke, H., Enger, T.A., Tolat, V., Scott, T.S., Taber, A.H., Sussman, R.J., Klein, W.A. und Wahaus, C.W.** (1996): IBM experiments in soft fails in computer electronics. IBM Journal of Research and Development, 40(1): Seiten 3–18. ISSN 0018-8646. doi:10.1147/rd.401.0003. (Zitiert auf Seite 3.)

[ZL79] **Ziegler, J. und Lanford, W.** (1979): Effect of cosmic rays on computer memories. Science, 206(4420): Seiten 776–778. ISSN 0036-8075. (Zitiert auf Seite 3.)

[ZP04] **Ziegler, J. und Puchner, H.** (2004): SER – History, Trends and Challenges: A guide for designing with Memory ICs. Cypress Semiconductor Corporation. (Zitiert auf den Seiten 3 und 4.)

Abstract

As technology scales, hardware designs for embedded systems offer more performance and parallelism for the price of being less reliable. Therefore, soft-error mitigation is one of the major challenges for safety-critical applications and systems. Soft-errors occur randomly and induce temporary malfunctions of the hardware, which have to be handled actively during execution.

The construction of reliable systems from unreliable components generally requires the introduction of redundancy, for example by means of structural replication. Besides adding costly hardware redundancy, virtually sacrificing the technology gain, software-based fault-tolerance offers a selective and resource-efficient alternative. Although being a proven technique in general, existing approaches still suffer from an incomplete coverage or significant residual error rates. *Single points of failure* and gaps in the intended redundancy domain cause the trouble. A majority voter is a case in point, which in itself cannot be redundant but is mandatory for redundancy schemes. Although these components are usually considered as relatively small and short in terms of execution time, their shape and quantity is highly application specific. Therefore, systematically improving the actual reliability by dint of software-based fault-tolerance tends to be very difficult.

In simple terms, software-based redundancy suffers from the causality dilemma of monitoring itself, keeping with Juvenal's critical demand: *But who can watch the watchmen?* (Satire 6, p. 347 sq.)

To solve this problem, this thesis develops the *Combined Redundancy* (CoRed) approach. This holistic software-based fault-tolerance scheme is based on redundant execution of the safety-critical application's processes. Moreover, it prevents redundancy shortages by selectively applying arithmetic coding techniques to cover any gaps in the replicated execution. Thereby, CoRed eliminates all remaining single points of failures and acts as *the watchman of the watchmen*; it thus offers the redundancy tools for a systematic improvement of the non-functional property *reliability* at the operating system level. An exhaustive experimental evaluation of CoRed's building blocks and the employed arithmetic coding – from the design to the machine-dependent implementation – satisfactorily shows the absence of errors and a significant improvement of the overall fault-detection performance in comparison to similar approaches. CoRed considers the input data acquisition and the output data distribution and thereby features a full input-to-output protection of safety-critical applications, which can be even extended over system boundaries.

Control applications usually possess designated reliability requirements because of their safety-related control function and close coupling with the physical environment. Therefore, the general feasibility of CoRed is demonstrated by the example of the mission-critical flight control of the I4*Copter* unmanned aerial vehicle. The approach can be applied transparently to the application developer in terms of functional aspects. However, the redundant execution of the application is inevitably interfering with the temporal properties assumed by the controllers design. This potentially has a negative impact on the control performance and thereby complicates the implementation of CoRed-based fault-tolerance in safety-critical control applications. With CoRed@Control, this thesis additionally explores an appropriate application-aware interface for CoRed in order to support and simplify the mapping of control activities to replicable processes of the underlying real-time operating system. This interface also enables the feedback of the changing temporal properties to the control application.